DOCTOR OF SCIENCES

Program understanding in database reverse engineering

Henrard, Jean

Award date:
2003

Awarding institution:
University of Namur

FUNDP
Institut d'Informatique
Rue Grandgagnage, 21
B-5000 Namur
Belgique

PROGRAM UNDERSTANDING IN DATABASE REVERSE ENGINEERING

Jean HENRARD

Thesis submitted for the degree of Doctor of Science
(Computer Science Option)

Jury : Professor Jean Fichefet, Institut d'informatique, FUNDP (President)
Professor Jean-Luc Hainaut, Institut d'informatique, FUNDP (Supervisor)
Doctor Rainer Koschke, Universität Stuttgart, Germany
Doctor Jean-Marc Petit, Université Blaise Pascal, Clermond-Ferrand, France
Professor Jean-Marie Jacquet, Institut d'informatique, FUNDP

August 2003

Acknowledgements

Even if my name is the only one to appears on the first page of this thesis, this work is a teamwork of more than ten years.

First of all I would like to express my gratitude to my promotor, Professor Jean-Luc Hainaut, for his endless advice about this work and for our numerous discussions about how to apply (transfer) our scientific "theories" to real life problems. I would also like to include all my colleagues of the database engineering laboratory both for the studious environment and the many asides we shared about computers, house renovation, volleyball, children, etc. Special thanks to Didier Roland, Jean-Marc Hick and Vincent Englebert for the co-development of the DB-MAIN CASE tool, with the wish that this collaboration may continue many years outside the university.

In this thesis, I have tried to apply the theoretical aspect of the methodology to real problems encountered in companies. That part of the thesis relies on many experiments conducted on database and source codes provided by several companies. I also want to thank these companies for allowing me to analyze some parts of their code and for contributing, without knowing it, to this thesis.

It would have been impossible to finish this thesis without the support of my family and friends during all those years. Thank you for listening to me while I was trying to explain what database reverse-engineering (archeology, recovering program's plans, understanding how a program works, etc.) is and for encouraging me even if they sometimes had problems figuring out what I was trying to achieve..

Special thanks to Marie-Berthe and Odile for their encouragement and patience during all those years. For sharing their husband and father with this endless thesis and computers.

Many thanks to all of you!

Jean

Abstract

For many years software engineering has primarily focused on the development of new systems and neglected maintenance and reengineering of legacy applications. Maintenance typically represents 70% of the cost during the life cycle of a system. In order to allow an efficient and safe maintenance of a legacy system, we need to reverse engineer it in order to reconstruct its missing or out-of-date documentation. In data-oriented applications the reverse engineering complexity can be broken down by considering that the database can be reverse engineered independently of the procedural components.

Database reverse engineering can be defined as the process of recovering the database's schema(s) of an application from database declaration text and program source code that use the data in order to understand their exact structure and meaning. A database reverse engineering methodology is broken down into three processes: project preparation, data structure extraction that recovers the database's logical schema and data structure conceptualization that interprets the logical schema in conceptual terms.

Data structure extraction is the most difficult process because it has to recover the database's complete structure from database declaration text and source code. When analyzing the source code, it quickly appears that program understanding techniques are needed. Program understanding is that software engineering domain that intends to gain knowledge about existing programs. We have adopted and applied some of the techniques of this domain (variable dependency graph, system dependency graph, program slicing) in order to help analysts to recover data structures and constraints from the source code.

In order to validate our methodology and program understanding techniques, we have developed tools to support them. Those tools have proved absolutely necessary to perform database reverse engineering of medium to larger applications in reasonable time and at reasonable cost. To cut down on the cost of large projects, we have stressed the need for automation to reduce the manual work of the analyst. Our experience with real size projects has taught us that the management aspects of a project are essential success factors. The management of a project comprises different aspects such as database reverse engineering explanation, cost evaluation and database reverse engineering result evaluation.

Table of Contents

CHAPTER 1 Introduction

1.1. General introduction

For many years software engineering has primarily focused on the development of new systems. Research and industrial efforts have concentrated on creating new and more efficient software development methodologies and processes to increase the quality of the applications, to decrease the time to market and to develop applications that really meet the demand of the customers. By focusing on those aspects, maintenance, which is one of the major (in time and cost) activities of the software cycle life, has been neglected. Maintenance typically represents 70% of the cost during the life cycle of the system [Leintz et al.-1980]. All large programs would undergo significant maintenance during their in-service phase. As changes are introduced into a system, its structure begins to deteriorate. Members of the original and intervening programming teams disperse. The documentation, if any, gradually becomes outdated. Such systems, called *legacy systems* contain business knowledge and mission-critical data. They become more and more difficult to change, correct, enhance but they need to evolve to follow evolution in business such as new laws, new business habits, new business opportunities, enterprise merges or absorption, new technologies, new software architectures, etc.

Newly created companies, without legacies, can just purchase or develop new software that take advantage of the latest technology like the web, client/server or open system standards (XML, CORBA). But older companies have to deal with their existing legacy systems in which considerable effort has been invested, and the replacement of which can prove highly risky.

Brodie [Brodie et al.-1995] defines a *legacy system* as "any system that significantly resists modifications and changes. Typically, a legacy system is big, with millions of lines of code, and more than 10 years old." Bennett [Bennet-1995] defines it as a "large software system that we don’t know how to cope with but that is vital to our organization".

Many legacy systems do not satisfy the flexibility and growth requirements of modern enterprises. They were built with focus on efficiency rather than on interoperability and maintenance. They are often badly documented. On the other hand, legacy systems are of great value because they incorporate important business knowledge and manage a vast amount of mission critical business data.

This resistance to change increases the cost of the maintenance. *Maintenance* is defined by Corbi [Corbi-1989] as "understanding and documenting existing systems; extending existing functions; adding new functions; finding and correcting bugs; answering questions for users and operations staff; rewriting, restructuring, converting, and purging software; managing the software of an operational system, and many other activities that go into running a successful software system".

To reduce the maintenance cost, that can be up to 70% of the total life cycle cost of the application, the enterprise has two solutions. The first one is to rebuild from scratch the entire system to meet the new requirements. This strategy, called *cold turkey* [Brodie et al.-1995], carries substantial risk of failure: such a project might require several years, during which the legacy system is likely to evolve.

Even if we want to replace the current system, we need to acquire a deep knowledge of the old one, because the new one needs to offer, at least, the same functionalities as the previous one and a lot of the current business rules are not explicitly described in some documents but are implemented in the system. For example, a given application obviously includes a function that computes mortgage rates, but nobody understands how it does it. Even if one decides to rewrite the application from scratch, an important part of the legacy system cannot be discarded, that is, its database. The list of customers, orders, products and unpaid invoices cannot be lost, but must be migrated to the new system. This migration clearly requires a deep understanding of the meaning of the data, their format and how they are stored in the database.

For all these reasons, it is impossible or very risky to throw away all the legacy systems and to build a new one from scratch.

Another strategy, called *chicken little*, is to migrate the legacy system by small incremental steps until the desired long-term objective is reached. The analysis and migration of a sub-component to new technologies while other legacy components remain unchanged is usually called reengineering.

To allow an efficient and safe maintenance of a legacy system, for which no precise and up-to-date documentation exists and of which the existing team does not master all the aspects, we need to reverse engineer it to reconstruct this missing documentation. The correct understanding of the system is a strong prerequisite before any modification. For instance, this knowledge is needed to evaluate the implication of the changes needed, to identify the parts of the system that will be affected and finally the cost of the modifications. If the programmer responsible for the change only has a partial view of the system, he cannot anticipate the implication of the change. The cost and the time needed to perform it exceed the forecast. When the change has been applied, some other parts of the system often do not work correctly anymore, so that the last change involves additional correction, and so forth.

Tilley [Tilley-1996] defines *reengineering* as "the systematic transformation of an existing system, or a part of it, into a new form to carry out quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer".

To achieve reengineering, we need to understand the structure and interrelationships of the system, through a process called *reverse engineering*. It has been defined by Chikofsky and Cross [Chikofsky-1990] as "the process to achieve understanding of the structure and interrelationships of a subject system. It is a goal of reverse engineering to create representations that document the subject and facilitate our understanding. As a process, reverse engineering can be applied to each of the three principal aspects of a system: data, process, and control".

In information systems, or data-oriented applications, i.e. in applications whose central component is a database or a set of permanent files, the complexity can be broken down by considering that files or databases can be reverse engineered (almost) independently of the procedural parts.

- The semantic gap between the so-called conceptual specifications and the physical implementation is most often narrower for data than for procedural code. For example, a COBOL field structure is easier to understand than a COBOL procedure.
- The permanent data structures are generally the most stable part of the applications.
- Even in very old applications, the semantic structures that underlie the file structures are mainly procedure-independent, though their physical structure is highly procedure-dependent.
- Reverse engineering the procedural part of an application is much easier when the semantic structure of the data has been elicited.

It is therefore much more efficient to first concentrate on the reverse engineering of the application's data components than to cope with the whole application. The reverse engineering of the data, called *database reverse engineering* (DBRE), is defined by Chikofsky and Cross [Chikofsky-1990] as "[a process that] concentrates on the data aspect of the system that is the organization. It is a collection of methods and tools to help an organization determine the structure, function, and meaning of its data" or by Hainaut [Hainaut et al.-1993a] "[the process of] recovering the schema(s) of the database of an application from DMS-DDL text and program source code that uses the data (and from any relevant source) in order to understand their exact structure and meaning".

The recovery of the lost information about the data structure of an information system (IS) is materialized by the creation of new system documentation. This documentation needs to be reconstructed because it has never existed or has been lost or has deteriorated during years of maintenance operations. Up-to-date and complete documentation is necessary for different purposes:

- *Maintenance*

  For efficient maintenance it is necessary to have a complete understanding of the current system to correctly evaluate the cost of a requested change. A global view of the system is also necessary to identify side effects of a given modification.
- *Database administration*

  A database administrator needs a good view of its database to evaluate the storage space needed and to forecast its evolution. The knowledge of the program that uses the data can help him to optimize (tune) the database to reach optimal response time.
- *Data conversion*

  To perform some data conversion, as was required for the year 2000 or Euro, the analyst needs to know the exact semantics of each field to know which one needs to be converted.
- *Data migration*

  A common problem, that requires an in-depth knowledge of an application, is the migration of the data from one database to another. This occurs when the hardware or software environments change, in order to publish the data on the web, to set up an ERP solution or to merge two information systems.
- *Data extraction*

  The knowledge of the data semantics is important when we want to use our current database to supply data to other information systems. Such extractions are used to populate datawarehouses or data mining tools to support strategic decisions. Data extraction is also necessary to support fashionable applications such as e-business and B2B.

- *Reuse*

  When users ask for a new function, it is important to know what the existing data structures are, in order to prevent the creation of new data structures and thus the creation of redundancy.

- *Evaluation of existing software*

  DBRE can also be used to assess the overall quality of software systems. A data structure with significant design flaws indicates poorly implemented software. Thus it can represent one of the evaluation criteria for a potential software product (homemade or vendor software).

1.2. Scope and motivation of the thesis

Experience quickly teaches us that recovering conceptual data structures can be much more complex than merely analyzing the data description language code of the database. *Data Description Language* (DDL) is a part of the *Database Management System*[1] (DMS) facilities intended to declare or build the data structures of the database. Untranslated data structures and constraints, non standard implementation approaches and techniques, old or esoteric DMS and ill-designed data structures are some of the common difficulties that the analyst encounters when trying to understand an existing database from operational components. Since the DDL code is no longer the unique information source, the analyst is forced to refer to other documents and system components that will prove to be more complex to analyze and less reliable. The most frequent sources of problems have been identified [Anderson-1996], [Blaha et al.-1995], [Hainaut et al.-1993a], [Petit-1996], [Premerlani et al.-1993] and can be classified as follows:

- *Weakness of the DMS models*

  The technical model provided by the DMS such as CODASYL-like systems, standard file managers and IMS DMS, can express only a small subset of the structures and constraints of the intended conceptual schema. In favorable situations, these discarded constructs are managed in procedural components of the application: programs, dialog procedures, trigger, etc. and can be recovered through procedural analysis.

- *Implicit structures*

  Such constructs have intentionally (or unintentionally) not been explicitly declared in the DDL specification of the database for optimization reasons, due to an oversight, or in order to be backwardly compatible with an older DMS. They have generally been implemented in the same way as the constructs discarded due to the weakness of the DMS models such as mentioned above.

- *Optimized structures*

  For technical reasons, such as time and/or space optimization, many database structures include non semantic constructs. In addition, redundant and unnormalized constructs are added to improve response time.

- *Awkward design*

  Not all databases were built by experienced designers. Novice and untrained developers, generally unaware of database theory and database methodology, often produce poor or even wrong structures.

1. We use the term Data Management System (DMS) which encompasses *DataBase Management System* (DBMS) and *File Management System* (FMS) such as COBOL file management libraries.

- *Obsolete constructs*

  Some parts of a database can be abandoned, and ignored by the current programs.
- *Cross-model influence*

  The professional background of designers can lead to very peculiar results. For instance, some relational databases are actually straightforward translations of IMS databases, of COBOL files or of spreadsheets [Blaha et al.-1995]. A CODASYL database carelessly translated into a relational database can explicitly introduce DB-keys (physical record identifiers) as columns in the new tables. Similarly, a COBOL record type looses its hierarchical field structure when translated into a single table.
- *Inconsistent standard*

  These systems have been developed and maintained for many years (several decades in some cases) and during which time the programming standards and methodologies, software and hardware have changed. The system is no longer homogeneous, but appears to be a collection of small subsystems, each one with its own characteristics. In some unfavorable cases, the system uses several programming languages or more than one DMS. For example, systems where COBOL indexed files coexist with a relational DMS are not infrequent.
- *Size of the system*

  Systems integrate more and more business processes and are developed over many years. As a result, such systems can be very large. For example, several million lines of code and more than 500 tables or record types is not exceptional. So, methods and techniques that seem fine for small projects become useless for medium or large ones.

Our experience showed us that most of the implicit structures and constraints are buried into the source code of the programs and that this code is often the most reliable place where such constraints can be found. Analyzing program source codes requires sophisticated techniques pertaining to the program understanding domain.

Müller [Müller-1996] defines *program understanding* (PU) or *program comprehension* as "the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for maintenance, evolution, and re-engineering purposes".

This introduces an interesting apparent paradox: DBRE is often intended to provide a better understanding of a data intensive program while program understanding contributes to DBRE. In the context of this thesis, the program understanding techniques that will be developed are not intended to build a mental model of the whole system, but to enhance our comprehension of the persistent data structure used by the application. In particular, we will study and develop three techniques, namely *programming pattern* analysis, *data flow* analysis and *program slicing*.

It quickly appears that systematic methodologies, relying on rigorous techniques and on powerful tools are necessary to successfully undergo such DBRE projects. In this thesis, we will present a generic DBRE methodology that can be applied to any DBRE projects independently of the DMS used. We have also developed tools that support this methodology, and that have been integrated into the DB-MAIN CASE environment [Hainaut et al.-1996b].

One of the major failure factors is the inability to manage and master the high volume of information, such as the source code of the programs and the data schemas. For instance, retrieving the foreign keys of a 250 record type database, totaling 10,000 fields, processed by 250,000 lines of

code, theoretically requires examining every pair of fields against each line of code, to check whether this pair represents a foreign key/unique key pattern. Though clever heuristics can radically reduce this search space, powerful tools are indispensable to automate, at least partially, the search for hidden constructs.

This thesis also explains why full automation of the whole DBRE process is often impossible. We have noticed that no two DBRE projects are identical, nor even similar. Major aspects such as the DMS (sometimes more than one), the programming language (quite often more than one), the specific way to code constraints and to name objects (those rules may change in a given application), can widely vary from one system to another one. For this reason, no predefined tool with hardwired techniques and heuristics can cope with this variety of situation. Hence the need for a programmable and extensible CASE environment through which analysts can develop new tools quickly.

1.3. The thesis

In this thesis, we will prove that program understanding techniques and tools significantly contribute to good quality DBRE for real size projects.

To prove the thesis, we will develop a DBRE methodology that covers all the DBRE life cycle and that can be applied to any kind of DBRE projects. In the context of this methodology, we will show that the program's source code is an up-to-date and complete source of information to recover the database structure.

Due to the size and complexity of programs, techniques are necessary to understand the programs. These techniques will be adapted to DBRE and tools will be developed to support the work of the analyst.

Finally to prove that our methodology and its supporting tools are adapted to real size projects, we will use them to solve real size case studies.

1.4. State of the art

Since 1980, a wide range of DBRE methods have been published. All of them consist of extracting the conceptual schema from an operational legacy system. The conceptual schema can be expressed in some variant of the entity-relationship model or of the object-oriented model (ODMG, UML, OMT).

Each method exhibits its own rules and heuristics, produces its own outputs and requires specific inputs and assumptions. Surprisingly, the amount of work on how to translate relational schemas into a conceptual model outweighs the work on the mapping from other physical model. There can be several reasons for this phenomenon. In the last few decades, a lot of work was done on the relational theory: (mathematical) formalization of the theory, methodology to design efficient relational databases, transformation of a conceptual schema into relational structures. No such theoretical background exists for network or hierarchical models and even less for standard files. Relational

database design is taught in all schools, universities and professional training database courses, so relational databases are supposed to be better designed than other (older) databases. Therefore, it is easier to recover the original design. Modern relational databases allow the implementation of almost all the constraints of the conceptual schema through foreign keys, primary keys, indexes (unique or not) and null values. The constraints that are not explicitly expressed in the DDL language are validated in the procedural part of the application using the data manipulation language (DML or queries). Complex queries can be expressed using this DML, therefore to recover the implicit constraints the analyst can only analyze the DML fragment without worrying about the other procedural part of the application. In some favorable (academic) cases the physical schema of the database contains all the constraints and thus does not require a program source code or data analysis.

For less powerful models, in particular the standard files, the physical model expresses very few constraints and makes it necessary to analyze the source code to recover the missing constraints. This code analysis is not an easy task as will be shown in this thesis.

Relational databases usually work on modern computers, nowadays these computers are powerful and disk accesses are very fast. Most of the relational database courses present the concept of normalization and encourage the building of normalized databases. On the other hand legacy databases are heavily optimized because when they were designed computers were expensive, and as a consequence, the designer was asked to save disk space and disk access. Moreover, the programmers were not trained in database design and did not use standard design methods.

All these reasons, and the fact that researchers prefer to work with and to study modern languages, explain why most of the DBRE methods address the problem of reverse engineering relational databases.

Various criteria can be used to classify the DBRE methods. We note in particular the following:

- *DMS supported*

  Most methods are specific to a DMS model.
- *Target model*

  Methods express the conceptual schema according to a conceptual model. This model can be a variant of the entity-relationship model (EER, ERC+, etc.) or some object-oriented model (ODMG, UML, OMT, etc.).
- *Prerequisites*

  Some methods have prerequisites regarding the database to be reverse engineered. The schema must be in third normal form (3NF), attributes' names must have some coherency (two attributes with the same name represent the same thing and two attributes with different names have different semantics) or there is no error in the data.
- *Thoroughness of domain semantics acquisition*

  Some methods require that the physical schema contains all the semantics (meaningful names, all the identifier are present) and the analyst knows the domain to interpret the missing information. Other methods analyze additional sources of information to retrieve the semantic missing in the physical schema.

- *Heuristics and techniques used*

  Description of the heuristics and techniques employed during the reverse engineering process. Some authors just explain some abstract heuristics, others give algorithms and tools are suggested or implemented.

- *Completeness and robustness*

  Legacy databases are real databases designed and maintained by real programmers. So these databases are rarely (if ever) designed according to the textbook rules and methods. Due to their maintenance, databases become inconsistent in object naming, some data structures are not used anymore, they are optimized, they exhibit design flaws, etc. Any reverse engineering process should be evaluated against these common characteristics. The completeness of a method is achieved if the process addresses the possible method flaws of the method. The robustness of a method represents how the method behaves when such flaws are present.

- *Automation/user interaction*

  To use a method in a real project, it is important to know what part of the method can be automated and when the interaction with the user is necessary.

- *Sources of information*

  There are a lot of information sources that can be analyzed (physical schema, program, data, documentation, user knowledge, etc.). Most of the methods only use some of them.

In the remainder of this section different methods will be briefly presented. They have been grouped according to the DMS they support.

1.4.1. Relational DMS

The first research efforts focused on transforming a relational schema with known primary keys and foreign keys to a conceptual schema. These approaches used information on tables, column names and primary keys. They did not specify the sources of information (the physical schema is given) and had heavy prerequisites on the physical schema (3NF, meaningful names, etc.). Some works try to recover an object-oriented conceptual schema and thus concentrate on the discovery of generalization hierarchies.

A trend in recent work is to put more emphasis on the information acquisition phase. Different sources of information for keys, foreign keys and inclusion dependencies are explored.

1.4.1.1. [Dumpala et al.- 1983]

[Dumpala et al.- 1983] is one of the earliest works where a description is given of how to map from the relational, network and hierarchical model to a conceptual model. This method requires, as input, logical schema in third normal form with information on (primary) keys and foreign keys. The output is an entity-relationship model with attributes and relationship types.

The methodology is presented as an algorithm to transform the logical model. This algorithm is based on the classification of the relations and the keys (primary keys, foreign keys). The different classes are defined, but no algorithm to automatize this classification is given, it must be done manually by the user.

1.4.1.2. [Navathe et al.-1987]

[Navathe et al.-1987] method is an enhanced version of the mapping algorithms of [Dumpala et al.-1983]. It requires, as input, the relations in third normal form and the key attributes which are used in more than one relation must have the same name throughout the schema. The output is a conceptual schema expressed in an enhanced version of the entity-relationship model, called Entity-Category-Relationship model (ECR), that introduces the concepts of subclasses and generalization hierarchies.

The methodology classifies relations and attributes. The conceptual schema is generated much in the same way as in [Dumpala et al.- 1983] except for the order of the phases and that the system interacts heavily with the user. Supertypes are created for entity types that have the same primary keys.

1.4.1.3. [Casanova et al.-1983]

[Casanova et al.-1983] method requires, as input, a relational schema with primary keys and foreign keys. This schema is expected to come from the user. The target conceptual schema is an entity-relationship model without complex objects nor generalization.

The method takes into consideration primary keys and foreign keys. Tables are split and merged (fold) into entity types that each represent one item in the conceptual schema. An algorithm that generates a schema is presented together with a formal proof that the generated schema is correct.

1.4.1.4. [Markowitz et al.-1990]

[Markowitz et al.-1990] method continues the formal approach taken in [Casanova et al.-1983]. It requires, as input, the relational schema, key dependencies and key-based inclusion dependencies, i.e. referential constraints. Relations are assumed to be in Boyce-Codd normal form. The output is an extended entity-relationship (EER) model.

The methodology has four steps. The first one transforms relational schema into a form appropriate for identifying EER object structures. The second step of the methodology examines relation-schemes, functional dependencies and inclusion dependencies obtained after the transformations in order to detect whether they satisfy a set of properties. The third step determines the type of object interactions, such as weak-entity-set and specialization, for each inclusion dependency. The fourth step derives a candidate EER schema using some mapping rules, finally the quality of generated EER schema is examined.

This method is very demanding on the input. The main contributions of the work are the independence from attribute names and the formalization of the mapping between schemas.

1.4.1.5. [Davis et al.-1987]

[Davis et al.-1987] method requires, as input, the relational schema in third normal form. It is not specified how this schema is to be obtained. The target model is the entity-relationship model with no complex objects nor generalization.

Table with a single attribute as key and tables with a key containing multiple attributes (if all the attributes of the key are a referential constraint or none of them) are translated into entity types. Tables with dangling keys (a part of the attribute of the key is a referential constraint) are translated into weak entity types with attached integrity constraints specifying that an instance of the weak entity type cannot exist without a corresponding instance of the strong entity type on which it is dependent. A table with a key that is a concatenation of primary keys of tables translated to entity type is translated into a many-to-many relationship type. Relationship types with an order higher than two are handled. If key attributes of a table that is translated into an entity type also appear as non-key attributes in another table translated into an entity type, then a many-to-one relationship type is created.

1.4.1.6. [Premerlani et al.-1993]

[Premerlani et al.-1993] method requires, as input, the data dictionary of the data description language source, the data of application and the analyst must know the semantics of the application. No assumptions are made on the relation and it copes with design optimization and bad design implementation. The method adopts the Object Modeling Technique (OMT) notation for modeling the conceptual schema.

Candidate keys are used rather than primary keys. Candidate keys are identified through an analysis of unique indexes, automatic scanning of data and from the user knowledge. In a modern system, the foreign keys may be present in the DMS. If not, foreign keys can be deduced by investigating matching names, domain and data types. Join clauses in views embody information on foreign keys as well as secondary indexes. Groups of foreign key attributes are searched for. A generalization hierarchy may complicate the identification of foreign keys. Inclusion analysis is retrieved from an analysis of the database extension.

A large number of data structures are searched for in the source schema and translated into the target schema. Classes that are linked with a one-to-one association indicate a generalization. Generalizations may have been implemented by pushing attributes of the supertype down to the subtypes. Mutually exclusive groups of attributes indicate that the subtypes have been pushed up to the supertype. Once a translation is generated, a large number of post-translation transformations are advocated, i.e. fusion of vertical fragmented classes.

This method recognizes that all the constraints are not expressed into the physical schema. Other sources of information, such as database extension, user knowledge, programs, need to be analyzed.

1.4.1.7. [Johannesson-1994]

[Johannesson-1994] method requires, as input, the relational schema, function dependencies and inclusion dependencies. The input relations must be in third normal form. The output is given in an entity-relationship model without complex objects but with generalization.

This method defines a set of transformations used to split relations that represent more than one object type. Different relations representing the same object type are collapsed into one single relation. The principle of the schema mapping algorithm is to map each relation into an object type and each inclusion dependency into either a generalization constraint or a relationship type. It is shown that a generated conceptual schema can represent as much information as the original relational schema. The method is based on the well-established concepts of relational database theory. It is

very complete, in terms of the description of the reverse engineering steps, but with the drawback of needing all keys and inclusion dependencies.

1.4.1.8. [Signore et al.-1994]

[Signore et al.-1994] method requires, as input, the relational schema, SQL and host language code. The output is an entity-relationship model defining complex multivalued attributes and generalization.

The methodology has three phases. The first phase identifies primary keys. If they are not present in the relational schema, they are deduced from the source code. The user must verify their correctness. The second phase is for the detection of the indicators of synonyms and referential constraints. These indicators are searched for in the source code (host language and SQL queries), if not present in the relational schema, and are then verified from integrity constraint enforcement in the code and by the user. The last phase is the conceptualization. Using the indicators found in the previous phases the conceptual model is derived.

It should be noted that this method is based on clues. The clues are adopted to cope with unusual implementation techniques, optimization choices, poor data definition language and code errors, among others. A possible tool to support this method is sketched.

1.4.1.9. [Chiang-1995]

[Chiang-1995] method requires, as input, a populated database. The relations are in third normal form and key attributes with the same domain must have the same name in all tables, since the names of key attributes are used to infer references between tables. If a non-third normal form structure cannot represent more than one entity type, it must be decomposed into detailed modeling structures manually. The input database contains any erroneous data instances in its key attributes. The output is an extended entity-relationship (EER) model with generalization hierarchies.

The methodology has three major steps. The first one classifies relations and attributes, based on the relational schema and its primary keys. In the second step, the inclusion dependencies (referential constraints) are searched for. The possible inclusion dependencies are automatically detected based on the identifiers and the attributes' names. Then, these inclusion dependencies are validated by querying the database. Finally, the EER components are identified using a list of rules.

It differs from the articles mentioned above in that it addresses the problem of information acquisition and takes inclusion dependencies into consideration. It produces an EER model that is semantically richer than the relational schema.

The article presents a tool, Knowledge Extraction System (KES), that supports the method. This tool is developed in Prolog and C and queries an ORACLE database.

1.4.1.10. [Ramanathan et al.-1996]

[Ramanathan et al.-1996] method requires, as input, a relational schema in third normal form with primary key and foreign keys. The output is a schema using the Object Modeling Technique (OMT) notation.

The methodology is broken down into three steps. The first identifies the tables that correspond to object-classes. The second step identifies the relationships. It defines three types of relationships: associations, generalizations/specializations and aggregations. The last step identifies the exact cardinalities of the associations. All the information required by the process comes mostly from the information on primary keys and foreign keys. The method thus provides a great potential for automation, but no tool is presented.

The article gives some hints on how to cope with non 3NF schema when this denormalization comes from an optimization process. It suggests how to resolve horizontal and vertical partitioning optimization.

1.4.1.11. [Petit-1996]

[Petit-1996] method requires, as input, the relational schema with unique and not null constraints, data instances and code (SQL queries). The output is an extended entity-relationship (EER) model. The methodology analyzes conditions in queries and views to recover the foreign keys and the functional dependencies. Then the schema is restructured to obtain a logical schema in the third normal form. Finally the schema is translated into a conceptual schema and a domain expert validates this schema.

In [Lopes et al.-2002], this method is extended by the analysis of the logical navigation to recover the inclusion dependencies. The logical navigation is the use of join columns, as an access path, to navigate in a relational database. The tool, DBA companion, that discovers such constructs, is presented.

1.4.1.12. Varlet [Jahnke et al.-1999], [Jahnke-1999]

The Varlet method requires, as input, all the available sources of information about a relational database: declaration of the relational database (SQL-DDL), procedural code (embedded SQL), extension of the database (the data), the documentation, etc. The output is an object-oriented conceptual schema (OMT).

This method, named Varlet, consists of two main phases: schema analysis and conceptual schema migration. In the schema analysis phase, the different parts of the database are analyzed to obtain a consistent and complete logical data structure. In the conceptual migration phase, this logical data structure is transformed into a conceptual schema.

The main contribution of this method is the use of a graphical language named Generic Fuzzy Reasoning Nets (GFRN) to represent the analyst's knowledge of the logical schema. GFRN specifications separate declarative knowledge from operational aspects. This approach allows uncertain and inconsistent analysis results to be dealt with.

A prototype CARE environment has been developed that uses GFRN specifications and includes a customizable front end.

1.4.1.13.[Alhaij et al.-2001]

[Alhaij et al.-2001] method requires, as input, the physical schema with primary keys and foreign keys and the user knowledge of the legacy database. The output is an object oriented schema, no specific model is specified. The method translates the relational schema into an intermediate model called Relational Intermediate Direct Graph (RIDG). The RIDG is a graph where each node represents a table and the edges show that there is a foreign key between two tables. The RIDG is transformed into classes.

The paper also presents an algorithm to migrate the data from the relational database to the object oriented database.

1.4.2. Hierarchical/network DMS

There are very few works on the DBRE of network and hierarchical databases and some of them are an extension of work on relational databases. They only rely on the physical schema and user knowledge and do not analyze the application programs nor the data.

1.4.2.1. [Dumpala et al.- 1983]

[Dumpala et al.- 1983] method requires, as input, the network or hierarchical physical schema. The output is an entity-relationship schema.

The method to recover a network schema takes recursive set types into account. The transformation is straightforward. Each record type is converted into an entity type, each set type into a relationship type with the same cardinality constraint and each recursive link into a recursive relationship type.

The method to recover a hierarchical schema is done by a two step process. First, all trees in the hierarchical schema are connected with respect to common, but eventually renamed record types. The result is a network-like schema. Second, all record types are replaced by entity type and all parent-child relationship types are replaced by relationship type.

1.4.2.2. [Fong et al. - 1993]

[Fong et al. - 1993] method requires, as input, the network or hierarchical physical schema and relies on user expertise in the domain. The output conceptual schema is an extended entity-relationship (EER) model.

The methodology to translate a network model into an EER model is divided into seven steps. 1) Each record type is mapped to an entity type. 2) Each set is mapped to a binary relationship type. The user has to determine its semantics (1-1, 1-N or isa and whether it is mandatory or not). 3) Record types in network schema can form a loop such that two different navigation paths can be used to access the same member record type. Either these two access paths carry different semantics and both need to be kept, or both carry the same semantics, where there are two access paths for some optimization and only one of them must be kept. The user must decide which access path needs not to be mapped to EER model. 4) Derive N-N and N-ary relationship types. 5) Derive

generalizations. 6) Map each record attributes to a corresponding entity type attribute. 7) Derive entity type keys.

The methodology to translate a hierarchical model into a conceptual one is the same except that step 3 is not applicable.

1.4.2.3. [Navathe et al.-1987]

[Navathe et al.-1987] method requires, as input, a hierarchical schema containing records and parent-child relationship. The assumption is made that each record has a unique key field or identifier. The output is an entity-category-relationship (ECR) model.

The methodology requires a preprocessing of the schema in order to simplify the structure. The duplicate records as well as all pointer records are eliminated. The mapping process itself is composed of seven steps. 1) Records representing an N-N relationship type are replaced by a relationship type. 2) The links, that represent an isa, are replaced by an isa hierarchy. 3) If the identifier of a record contains the identifier of its parent record, then this record is said to be weak. 4) All the records that represent an N-ary relationship type are transformed into an N-ary relationship type. Steps 5 and 6 find different kinds of isa hierarchy (depending on whether the keys of the records are the same or not). 7) All the remaining records are transformed into entity types and all the remaining links are transformed into relationship types.

1.4.2.4. [Winans et al. - 1990]

[Winans et al. - 1990] method requires, as input, an IMS DBD and explicit behavior (the program that updates the database). The output is an entity-relationship model.

This methodology collects the information present in the IMS DBD. Then the segments are translated into entity types and relationship types are created to represent the parent-child links. The fields declared within the IMS DBD are translated into attributes of the appropriate entity type, sequence fields are translated into identifiers. Finally, all the entity types are examined to determine whether or not they can be combined. The behavior is used to add new constraints.

1.4.2.5. [Tangorra et al.-1995]

[Tangorra et al.-1995] method requires, as input, a hierarchical physical schema. The output is an entity-relationship schema.

This methodology is broken down into four steps. In the first, each segment is translated into an entity type with corresponding attributes and identifier. In the second step, the links in the hierarchical schema are translated into relationship types. Through a global entity-relationship schema analysis, the third step recovers N-ary relationship types and N-N relationship types. Finally the schema is restructured to make it more readable and meaningful.

1.4.3. Standard files DMS

In the database reverse engineering community, most research is on the topic of reverse engineering from a database (relational, network or hierarchical) into a conceptual (ER or OO) model. Less research has been devoted to reverse engineering applications using a file system for persistent storage.

1.4.3.1. [Davis et al. - 1985]

[Davis et al. - 1985] method requires, as input, a physical schema of the conventional files with file reference keys (kind of foreign keys). The output schema is an entity-relationship model.

The methodology translates the conventional file schema into the Current Physical Model (CPM). This CPM is composed of physical data units (PDU). PDU are created from record types, a composite data item (connected to its parent PDU), repeating groups (connected to its parent PDU) and an optional data item (REDEFINE, connected to its parent PDU). Each PDU is assigned to a data item in the COBOL data structure as its key. A set of PDU reference keys between PDU is derived based on the position of the PDU within the record type and partly given by the user.

The CPM is then translated into the conceptual schema. Each PDU is translated into an entity type, the major task being to locate the relationship types.

An experimental expert system, AUGUST, assists database designers in the translation of a conventional file system into a conceptual model.

1.4.3.2. [Anderson-1996]

[Anderson-1996] method requires, as input, COBOL source files. The output is an extended entity-relationship model, called ERC+, that extends the entity-relationship model with multi-instantiation, multivalued and complex objects.

This methodology recovers the third normal form schema of the database and then translates it into an ERC+ schema. To recover the former schema, the references between records and the dependencies must be identified. This is done by the construction of sets of variables that share the same value, called spreading sets. The fields of record types that are used to represent references are called anchors.

1.4.4. Generic methods

Methods are called generic if they are not specific to a particular DMS.

1.4.4.1. [Hainaut-1981]

[Hainaut-1981] method requires, as input, a logical schema in which all the constraints of the database are expressed. The output is a binary entity-relationship model.

This paper does not explicitly describe a DBRE method. It describes a set of reversible transformations from a binary entity-relationship model toward a DMS model. It uses relational and total DMS models to illustrate the examples. It is shown that the proposed transformations are reversible so it is possible to transform from an entity-relationship model to DMS model and the entity-relationship model can be deduced back from the DMS model. Each transformation is formally described and can be easily done automatically.

1.4.4.2. DB-MAIN [Hainaut-1991]

The DB-MAIN method requires, as input, all the information about the legacy database such as the DDL, the source code, the data, the documentation, etc. It produces two outputs. The logical schema, that is the schema the programmer must understand to be able to modify the legacy database and the programs that modify the data. The second output is the conceptual schema as an entity-relationship schema.

The method does not make any assumption about the language used by the legacy database nor its level of optimization. The method is decomposed into two main processes. The first one, the data structure extraction, produces the logical schema through the analysis of all the sources of information available. It contains two sub-processes, the DDL analysis, that extracts all the constraints declared in the DDL code, and the schema refinement, that extracts all the other constraints through the analysis of the other sources of information. The second main process is the data structure conceptualization. This process transforms the logical schema into the conceptual schema. It is made up of three sub-processes. The de-optimization removes the optimizations of the schema. The untranslation transforms the constructions specific to the DMS model into their conceptual equivalent. Finally the conceptual normalization gives to the final schema the desired quality of a conceptual schema.

This method is applied to standard files, hierarchical DMS, network DMS and relational DMS. There is a tool (DB-MAIN) that support the method.

1.4.4.3. MeRCI [Comyn et al.-1996]

The MeRCI method is not specific to a DMS model, but it is explained using the relational model. This method requires, as input, the source code of the application: the declaration of the database (SQL-DDL) and the procedural code of the application (with embedded SQL). The output is an extended entity-relationship (EER) model with complex attributes and generalization/specialization.

The name of this method is MeRCI (Méthode de RétroConception Intelligente). This method is made up of five steps. The first extracts the physical schema from the data dictionary, the DDL, the views. The second step applies a set of physical reverse engineering rules to "deoptimize" the physical schema. To detect the optimization they examine the DDL, SQL queries and the database extension. The third step identifies entity types, relationship types and cardinalities through the analysis of the embedded SQL, synonyms and references constraints. They enrich the indicators' rules proposed by [Signore et al.-1994]. The fourth step recovers the generalization/specification through the analysis of the queries and the data. The last step is the conceptualization that identifies multivalued attributes, entity types and relationship types.

[Akoka et al.-1998] presents an expert system that implements the MeRCI method.

1.4.5. Others

1.4.5.1. MeRCI-M [Akoka et al.-1999]

The MeRCI-M method is an extension of the MeRCI method [Akoka et al.-1998] to the reverse engineering of a datawarehouse. It requires, as input, the logical schema of the datawarehouse (with the different dimensions, variables and relations), it questions the users and it queries the data. While the MeRCI method foresees the extraction of the physical schema and its deoptimization, this is not yet implemented into MeRCI-M. Only the conceptualization process is presented.

The output is an extended entity-relationship model. The reverse engineering process is expressed as a set of rules that transform the datawarehouse logical schema into its conceptual counter part. These rules can be classified into three categories:

- The rules of "presumption" which emit a suspicion regarding the presence of a concept.
- The rules of "consolidation" which, by the exploration of a second source of information, reinforce the suspicion.
- The rules of "confirmation" which establish the presence of the concept.

1.4.5.2. FORE [Lee et al.-2000]

The FORE method, called form driven object-oriented reverse engineering (FORE), captures the form's semantics to derive a conceptual object-oriented model. The conceptual model is expressed via the object model on the CRC (class, responsibilities and collaborators) cards. A mapping is proposed to produce an UML model (class and sequence diagram).

It states that most of the knowledge about business applications can be compiled from business forms and the user's interaction with a legacy system. This methodology is divided into two major processes. The first, the form analysis, extracts the form structure and the user interaction with the legacy application. The second process, the reverse engineering, recovers the conceptual schema using the knowledge captured from the previous phase. The reverse engineering is divided into four phases: form object slicing, object structure modeling, scenario design and model integration. The form object slicing splits the form knowledge into semantic units according to the input types. The objective of the object structure modeling phase is to identify objects from the result of the previous phase. The scenario design phase produces an object process action scenario diagram. Finally the model integration phase integrates the models.

1.4.5.3. [Tan et al.-1997]

[Tan et al.-1997] method recovers inclusion dependency by the analysis of the programs. It starts from the observation that most (or all) of the inclusion dependencies in existing databases are enforced in the programs which update the databases.

The proposed theory is based on the fact that the records in a database result from the executions of the programs which update the database. Therefore dependencies can be inferred from the ways in which the program updates the database. The proposed approach is divided into two processes, recovery and validation. During recovery, inclusion dependencies are detected from dataflows between a record manipulation instruction (read, write, update) and an output instruction (write,

update) of another record. Validation checks if the inclusion dependencies found during the recovery process are not violated by another program.

There are no tools to support this method but a formal proof of its correctness is given.

1.4.6. Summary

This section presents a table summarizing the characteristics of the methods presented. The *DMS* column specifies the DMS supported by the method (*R*=relational, *N*=network, *H*=hierarchical and *gen*=generic model). The *target* column gives the target conceptual model. The *prerequisites* column enumerates the method prerequisites regarding the database to be reverse engineered. The *sem. recovery* column gives the technique used to recover the semantics (if the semantics is recovered) that is not expressed in the physical schema. The *heuristic* column lists the heuristics used by the method to recover the conceptual schema. *Transfo* means that the conceptual schema is obtained by the transformation of the physical schema. Most of the authors do not use the term transformation but the term rewriting, they create the conceptual schema by adding to this schema the objects that correspond to the elements of the physical schema. For simplicity, we have grouped this in the term transformation. The *tool* column gives the name of the tool that supports the method if it exists. And finally the *source info.* column is the list of information sources used by the method.

	DMS	Target	Prere-quisites	Sem. recovery	Heuristics	Tool	Source info
[Dumpala et al.- 1983]	R, N, H	E/R	Id, FK	-	transfo	suggested	phys sch
[Navathe et al.-1987]	R, H	ECR	3NF, name	-	transfo	-	phys sch
[Casanova et al.-1983]	R	E/R	Id, FK	-	transfo	-	phys sch
[Markowitz et al.-1990]	R	E/R	Id, FK	-	transfo	-	phys sch
[Davis et al.-1987]	R	E/R	3NF	-	-	-	phys sch
[Premerlani et al.-1993]	R	OMT	-	user knowl-edge	sch and data analysis, transfo	OM Tool	DDL, data, domain knowledge
[Johannesson-1994]	R	E/R	3NF with incl. depen-dencies	-	transfo	-	phys sch
[Signore et al.-1994]	R	E/R	-	code analysis	code analy-sis, Transfo	suggested	DDL, SQL, code
[Chiang-1995]	R	EER	3NF, name, no error in data	sch analysis (name)	sch and data analysis, transfo	expert system	phys sch, data, domain knowledge
[Ramanathan et al.-1996]	R	OMT	3NF, no opti-mization, Id, FK	-	transfo	suggested	phys sch
[Petit-1996]	R	EER	-	query analysis	query analy-sis, Transfo	DBA companion	DDL, DML, data
Varlet [Jahnke et al.-1999], [Jahnke-1999]	R	OMT	-	code and data analysis	sch analysis, transfo	prototype	DDL, DML, data, doc, ...

	DMS	Target	Prere-quisites	Sem. recovery	Heuristics	Tool	Source info
[Alhaij et al.-2001]	R	OMT	Id, FK	-	transfo	-	phys sch, domain knowledge
[Fong et al. - 1993]	N, H	E/R	-	-	transfo	-	phys sch
[Winans et al. - 1990]	IMS	E/R	-	-	transfo	-	DDL, code
[Tangorra et al.-1995]	H	E/R	-	sch analysis	transfo	-	phys sch
[Davis et al. - 1985]	F	E/R	FK	-	transfo	August	phys sch
[Anderson-1996]	F	ERC+	-	dataflow anal-ysis	code analy-sis, transfo	prototype	code
[Hainaut-1981]	Gen	ER	logical sch	-	transfo	-	log sch
DB-MAIN [Hainaut-1991]	Gen	ER	-	code and data analysis	code and data analysis, transfo	DB-MAIN	DDL, code, data, domain knowledge
MeRCI [Comyn et al.-1996]	Gen	EER	-	code and data analysis	code and data analysis, transfo	prototype	DDL, DML, data

1.5. Outline of the thesis

This thesis is organized as follows. The next chapter describes a generic data structure model according to which schemas of different abstraction levels and according to the most common paradigms can be described precisely. It explains how the various concepts of physical, logical and conceptual schema can be expressed in this model. Then, schema transformation operators that can model inter-schema transitions are presented.

Chapter 3 introduces a generic database reverse engineering (DBRE) methodology as the reverse of the database forward engineering. This method is divided in three processes (project preparation, data structure extraction, data structure conceptualization) and produces two schemas (logical schema and conceptual schema). The project preparation identifies the information available and the resource needed. The data structure extraction process aims at rebuilding the logical schema of the database through the analysis of all the information sources available. This schema is the view the programmer has (or must have) of the database to correctly write or modify any program that accesses the database. The data structure conceptualization process transforms the logical schema into a conceptual schema.

Chapter 4 describes in detail the data structure extraction process. This process is divided in four steps. The first, the DDL code analysis, extracts from the data description language script the explicit structures and constraints in order to produce the raw physical schema. If more than one raw physical schemas exist, the physical integration step integrates them into a single schema. The schema refinement step enriches the integrated schema with explicit constraints revealed by the analysis of the source code, the data, etc. Finally, the schema cleaning step discards the physical constructs that are no longer needed. The main constraints that are searched for are described as well as the elicitation techniques that are used to recover the constraints during the schema refinement steps.

Chapter 5 shows that implicit constraints can be elicited through the analysis of the source code. It is also stated that the source code is one of the most accurate and up-to-date sources of information for the recovery the implicit constraints. Due to the difficulty and cost of source code analysis, the analyst must have program understanding techniques and tools.

In chapter 6, program understanding techniques are adapted to help the analyst in his task of retrieving the implicit constraints that are implemented in the programs. Five program understanding techniques used to retrieve data constraints are presented: pattern matching, variable dependency graph, program slicing, system dependency graph and graphical visualization.

Chapter 7 explores how the program understanding techniques presented in the previous chapter can be used to retrieve the implicit constraints and structures presented in chapter 4. For the constraints that are generally searched for, it is explained how the program understanding techniques can be used and some hints are given on how the constraint discovery can be automated.

The DB-MAIN CASE tool is presented in chapter 8. To introduce the functionalities offered by DB-MAIN, the limits of the current reverse engineering CASE (CARE) tools are described and the requirements of an ideal CARE tool are given. It is shown how DB-MAIN fulfills these requirements and the CARE functions are explained.

Chapter 9 contains three case studies that implement the techniques described in this thesis. These case studies are small case studies that show different aspects of DBRE projects. The first case study is a COBOL program and stores the data into files. This case study shows how such a project can be done manually. The second case study is the same program as the previous one, but this time most of the work is done automatically. The last case study is a COBOL program with embedded SQL, to show how to cope with embedded code.

Chapter 10 tackles a less technical, but very important, aspect of DBRE projects namely management and planning issues. DBRE projects are risky and costly projects that do not bring new functionalities to the applications. Thus it is important for such projects to be supported by the managers of the company and not only by the technical team. Another difficulty of DBRE projects is that reverse engineering is not a well known process, so at the beginning of a new project it is necessary to explain to the managers and to the technical team what DBRE is. The cost of such a project must be evaluated. This cost evaluation utilizes many parameters such as the size of the database, the size of the program, the programming language, the analyst's experience, etc. The automation of the different steps is presented as a way to decrease the cost of the DBRE.

CHAPTER 2 *Data schema specification*

This chapter describes a generic data structure model according to which schemas can be described. This model can represent schema at different abstraction levels and of the most common paradigms. It explains how the various concepts of physical, logical and conceptual schema can be expressed in this model. Then, schema transformation operators that can model inter-schema transitions are presented.

2.1. Introduction

Database reverse engineering mainly deals with schema extraction, analysis and transformation. In the same way as for any other database engineering process, it must rely on a model that contains a rich set of data structures. This model must be able to describe data structures at different levels of abstraction, ranging from physical to conceptual, and according to various modeling paradigms. During DBRE, it is common to have parts of a schema of different levels of abstraction or of different modeling paradigm. The chosen model must be able to represent such situations.

In addition, statically describing data structures is insufficient. We must be able to describe how a schema has evolved into another one. For instance, a physical schema leads to a logical schema, which in turn is translated into a conceptual schema. These transitions, which form the basic of DBRE, can be explained in a systematic way through the concept of schema transformation.

2.2. A wide-spectrum specification model

In database development methodologies, the complexity is broken down by considering three abstraction levels. The engineering requirements are distributed among these levels, ranging from correctness to efficiency. At the conceptual level, the designer produces a technology-independent specification of the information, expressed as a *conceptual schema*, relying on an ad hoc formalism, called a *conceptual model*. At the *logical* level, the information is expressed in a model for which a technology exists. For instance, the required information is organized in a relational or object-

oriented logical schema. Since reverse engineering is concerned with legacy systems, we will also consider network, hierarchical, relational, indexed files, shallow, inverted files *logical schemas*. While a logical schema is based on a family of DMS models, a *physical schema* is dedicated to a specific DMS. In addition to logical constructs, it includes technical specifications that govern data storage, access mechanisms, concurrency protocols or recovery parameters. We will talk about network logical schema and about, say, IDMS physical schema.

Schemas manipulated during DBRE usually contain parts in different levels of abstraction or model abstraction. For example, the reverse engineered application can use COBOL files and an SQL database together. Both data structures must be displayed on the same schema, to represent relations between the data stored in the COBOL files and the one stored in the SQL database. For those reasons, specific formalisms cannot be adopted for each of the abstraction levels. Instead, the discussion will be based on a generic model that can easily be specialized in specific models. For instance, this model hierarchy can be translated into UML/relational/oracle 9i, into ERA/CODASYL/IDMS or into ORM/OO/Oracle ORDBMS design methodologies. These models are derived from a unique model. The formal basis of this generic model has been developed in [Hainaut-1989].

For the ease of the presentation, the generic model will be presented according to the three levels of abstraction (conceptual, logical and physical) in three different sections.

2.2.1. Conceptual specifications

In this thesis, all the conceptual schemas will be expressed using an extended entity-relationship model.

A conceptual schema mainly specifies entity types (or objects classes), relationship types and attributes. *Entity type* can be organized in *isa hierarchies* (organizing supertype and sub-types), the hierarchy can be total and/or disjoint. *Total* (T) means that a supertype must be specialized in at least one sub-type and *Disjoint* (D) means that a supertype can be specialized in at most one sub-type. A isa hierarchy that is both total and disjoint is called it a *partition* (P). An entity type can inherit from more that one entity type (supertype), this is called *multiple-inheritance*.

Entity types can comprise *attributes* that can be atomic or compound. The source value set of an *atomic attribute* can be a basic domain (e.g., numeric, boolean, character, time, etc.), a user-defined domain (e.g., VAT-number, address, URL, etc.) or an object class (in some OO models). A *compound attribute* is an attribute that is decomposed into at least one attribute (atomic or compound). Attributes are characterized with a *cardinality* constraint [i-j] stating how many values can be associated with a parent instance (default is [1-1]). *i* is the *minimum cardinality*, i.e. the minimum number of values that need to be associated and *j* is the *maximum cardinality*, i.e. the maximum number of values that can be associated. The following constraints on the values of *i* and *j* must be satisfied

$$0 \leq i < \mathrm{N} \text{ and } 0 < j \leq \mathrm{N} \text{ and } i \leq j$$

with N representing the infinity. If the minimum cardinality is 1, the attribute is said to be *mandatory* and if it is 0, the attribute is, then, said to be *optional*. If the maximum cardinality is equal to 1, the attribute is *single-valued* and if it is greater than one, it is said *multivalued*.

A *relationship type* has two or more roles. Each *role* also has a cardinality constraint [i-j] that states in how many relationships an entity will appear with this role. A relationship type with two roles is called *binary*, while a relationship type with n>2 roles is generally called *n-ary*. Roles can be *multi-*

domain, i.e. a role is connected to more than one entity type. Relationship types can also comprise attributes.

Entity types and relationship types can have constraints (such as identifiers), made up of attributes and/or remote roles. Those constraints are expressed through the concept of *groups*. A group is made up of components, which are attributes, roles and/or other groups. A group represents a construct attached to a parent object (entity type, relationship type or compound attribute). It is used to represent concepts such as identifiers, exclusive or coexistence constraints:

- *Primary identifier* (id)

  The components of the group make up the main identifier of the parent object. A parent object can have at most one primary id; all its components are mandatory.
- *secondary identifier* (id')

  The components of the group make up a secondary identifier of the parent object. A parent object can have any number of secondary identifier.
- *coexistence* (coex)

  The components of the group must be simultaneously present or absent for any instance of the parent object.
- *exclusive* (excl)

  Among the components of the group at most one must be present for any instance of the parent object.
- at-least-1 (at-lst-1)

  Among the components of the group, at least one must be present for any instance of the parent object.
- exactly-1 (exact-1)

  Among the components of the group, one and only one must be present for any instance of the parent object (= exclusive + at-least-1).

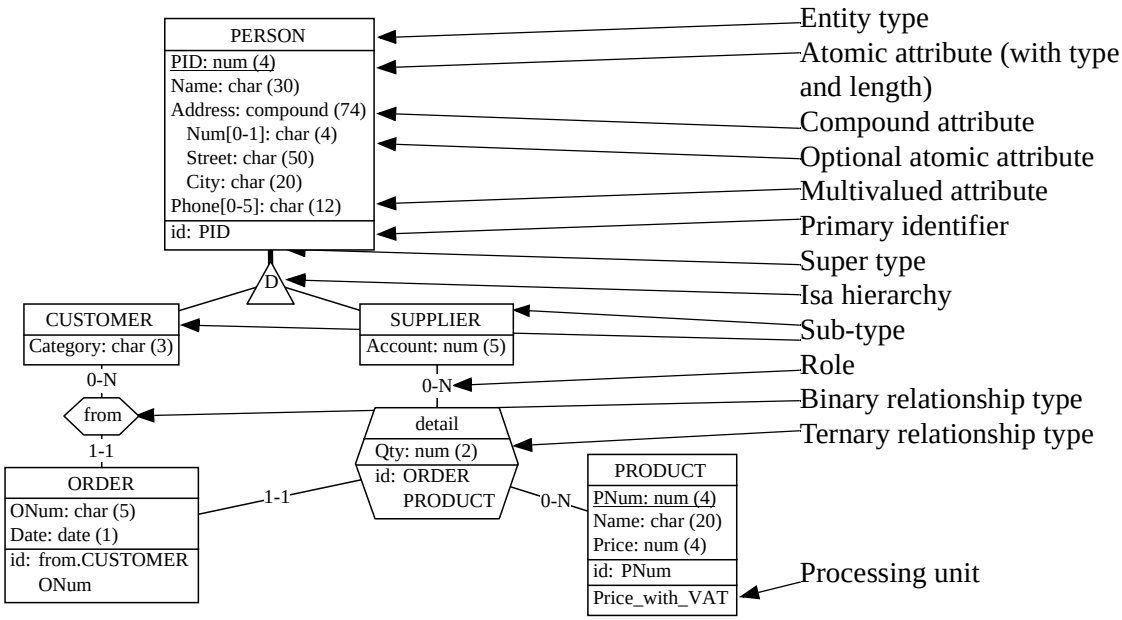


FIGURE 1. Example of a conceptual schema.

A *processing unit* is any dynamic or logical component of the described system that can be associated with a schema, an entity type or a relationship type. For instance, a process, a stored procedure, a program, a trigger, a business rule or a method can each be represented by a processing unit.

Some of these constructs are illustrated in figure 1. This schema includes entity types *PERSON*, *CUSTOMER*, *SUPPLIER*, *ORDER* and *PRODUCT*. *PERSON* has two disjoint (D) sub-types, *CUSTOMER* and *SUPPLIER*. Relationship type *from* is binary while *detail* is ternary. Each *ORDER* entity appears in exactly one *from* relationship (cardinality [1-1]). Entity types and relationship types have attributes. For entity type *PERSON*, attribute *Name* is atomic, single-valued and mandatory. *Address* is a compound attribute. Its component *Num* is atomic, single-valued and optional (cardinality [0-1]). *Phone* is multivalued and optional (cardinality [0-5]): there are from 0 to 5 values per entity.

{*PID*} is the primary identifier of *PERSON*. The identifier of *ORDER* is made of the role of the external entity type *from.customer* and of local attribute *ONum* ({*from.customer, ONum}*). The identifier of *detail* states that the relationship type detail is identified by its *ORDER* and its *PRODUCT*. There cannot exist more than one *detail* relationship with the same *ORDER* and *PRODUCT* entity types.

2.2.2. Logical specifications

A logical schema comprises data structure definitions according to one of the commonly used families of models: relational model, network model (CODASYL DBTG), hierarchical model (IMS), shallow model (TOTAL, IMAGE), inverted file model (DATACOM/DB), standard file (COBOL, C, RPG, BASIC), object-oriented model (Versant), object-relational (Oracle) to mention the most important ones.

The logical specification use the same concepts, of the generic model, as the one presented in the conceptual specification. Some of the concepts presented in the conceptual specification are not allowed in some logical models. For example relationship types, compound attributes, multivalued attributes, isa hierarchies are not allowed in the relational model. These concepts need to be expressed by equivalent constructs allowed in the model. Some parts of the conceptual schema cannot be translated into the logical one because the logical model is more restrictive. Those constraints are noted as a textual annotation below the schema.

New constructs that appear at this level are:

- *Special kinds of multivalued attribute*

  In the conceptual specification a multivalued attribute represents sets of values, i.e. unstructured collections of distinct values. At the logical level there is six categories to implement a multivalued attribute:

  - *Set*: unstructured collection of distinct elements (default).
  - *Bag*: unstructured collection of (not necessarily distinct) elements.
  - *Unique list*: sequenced collection of distinct elements.
  - *List*: sequenced collection of (not necessarily distinct) elements.
  - *Unique array*: indexed sequence of cells that can each contain an element. The elements are distinct.
  - *Array*: indexed sequence of cells that can each contain an element.

- *Reference constraint* (ref)

  This is an inter-group constraint where the origin group is the reference group and the target group is the referenced group. The referenced group must be an identifier (primary or secondary). Each instance of the first group must be an instance of the second group.

- *Inclusion constraint* (incl)

  This is an inter-group constraint where each instance of the first group must be an instance of the second group; since the second group does not need to be an identifier, the inclusion constraint is a generalization of the referential constraint.

- *Equality constraint* (equ)

  As the reference constraint, in addition, an inclusion constraint is defined from the second group to the first one.

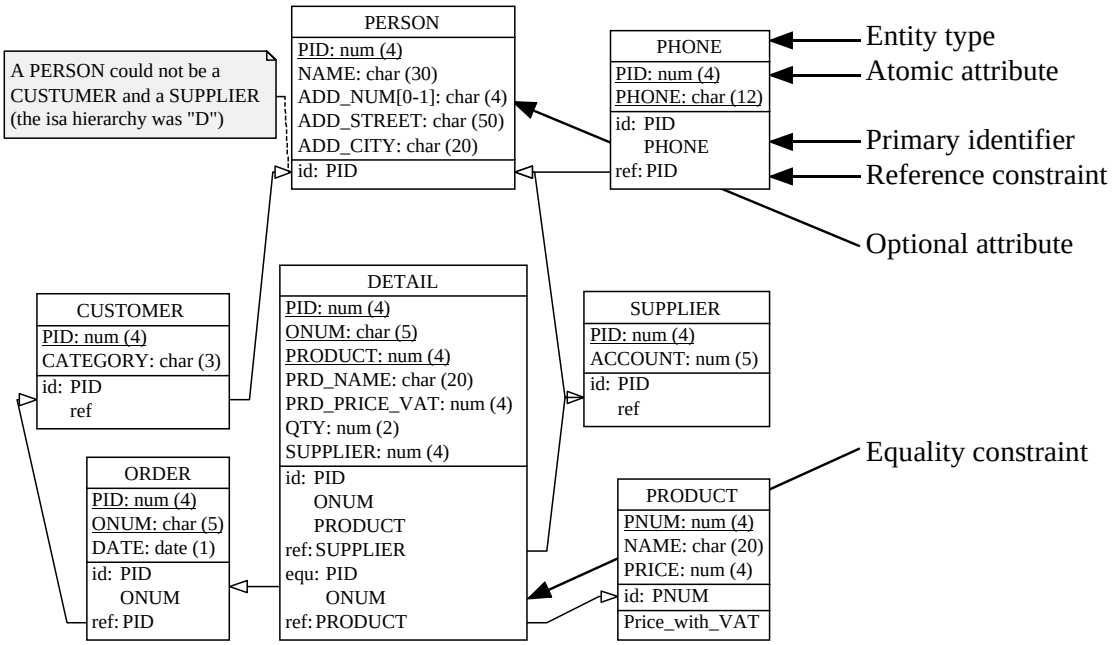


FIGURE 2. Logical schema, this relational schema is an approximate translation of the schema of figure 1.

For instance, the schema of figure 2 can be considered as a possible relational translation of the conceptual schema of figure 1. In this schema, all the relationship types and compound and multivalued attributes have been transformed to obtain a schema with only entity types, mandatory or optional atomic single-valued attributes, primary identifiers, referential constraints and equality constraints. Some constraints of the conceptual schema could not be expressed in this schema and are kept as annotation; such as the disjoint property of the isa hierarchy.

Similarly, it is possible to design a network, hierarchical or object-oriented equivalent schema. Since we want to discuss reverse engineering problems independently of the DMS model, we will use general terms such as entity type, relationship type and attribute. For a specific model, these terms will be translated into the specific terminology of the model. For instance, entity type will be called *table* in relational schemas, *segment types* in hierarchical schemas, and *data sets* in shallow schemas. A relationship type will be read *set type* in the network model, *access path* in the shallow model and *parent-child relationship* in the hierarchical model.

The level of details of the logical schema must be sufficient to the programmer to write the programs that use those data structures. So the needed level of detail differs from one model to the other. For example, to write a program that uses a relational DMS, the programmer only needs to know the entity type and attributes names, the identifiers and the referential constraints. On the other end, if the application uses standard files, he needs to know the name of the entity types and attributes, the identifiers and the foreign keys, like for the relational one. But he also needs to know the name of the collections and the access keys.

2.2.3. Physical specifications

Finally, physical specifications can be expressed through a physical schema. Due to the large variety of DMS-dependent features, it is not easy to propose a general model of technical constructs. As far as reverse engineering is concerned, two essential concepts will be considered that may bring structural or semantic hints:

- *Collection*

  Collection is an abstraction of file, data set, tablespace, dbspace and any record repository in which data is permanently stored.

- *Access key* (acc)

  Access key is a group that represents any path providing a fast and selective access to entity types that satisfy a definite criterion. Such as indexes, indexed set (DBTG), access path, hash files, inverted files, indexed sequential organizations all are concrete instances of the concept of access key.

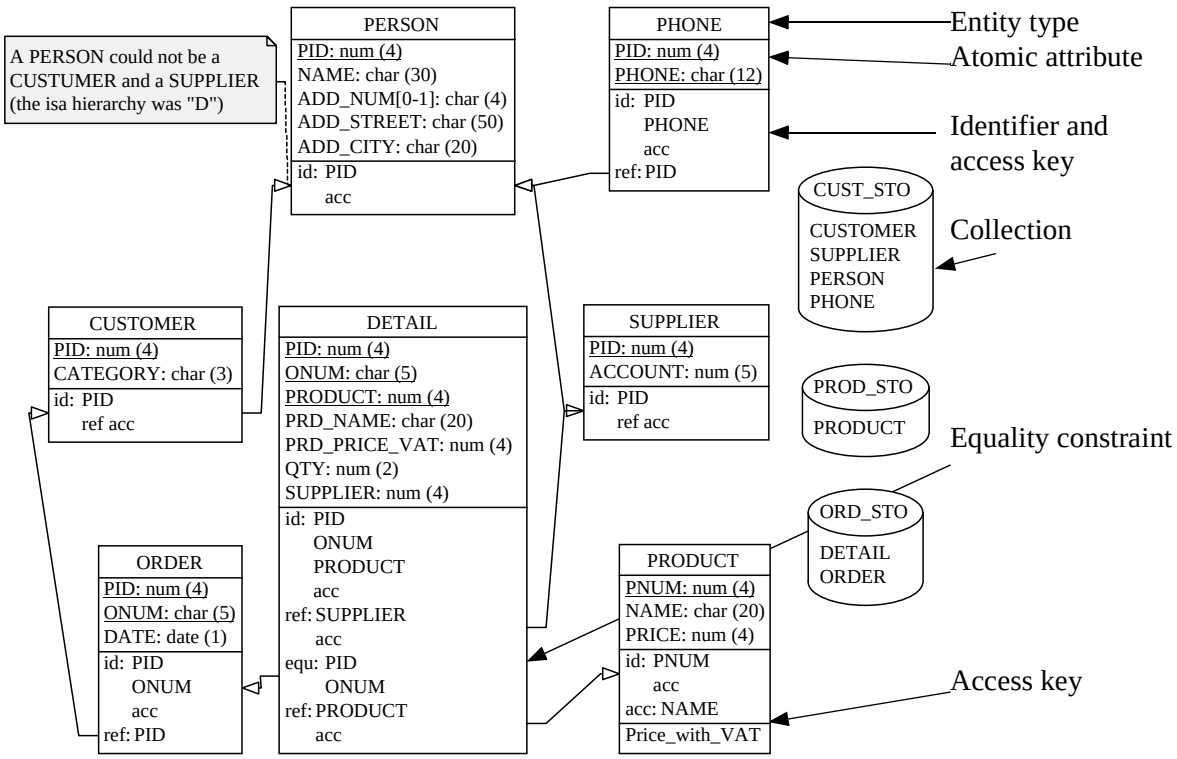


FIGURE 3. Physical schema. This schema grossly derives from the schema of figure 2.

These constructs have been given a graphical representation (figure 3). In database design and development, other physical constructs can be of interest, such as page size, extent size, file size, various fill factors, index technology, physical device and site assignments, etc. They will be ignored in this presentation.

Figure 3 schema is made up of seven physical entity types and three collections. Collection *CUS_STO* will store entity types *CUSTOMER*, *SUPPLIER*, *PERSON* and *PHONE*. The primary identifiers are supported by an access key, denoted by the symbol *acc*. An access key is also associated with some foreign keys and another one is defined on plain column {*NAME*} (to optimize the access to a product through its name).

Other technical details are not shown in this graphical representation, such as record clusters, physical column sizes and coding schemes, page lengths, buffer management, index technologies and parameters, etc. But they are stored as textual annotations.

The physical specification adds to the logical one all the information needed to implement efficiently the database using a given DMS. The physical specification has particular characteristics regarding its syntax, the constraints that can be expressed and its performances.

2.2.4. Different levels of abstraction and different paradigms

DBRE is an exploratory process that has to deal with existing (legacy) systems. The exploratory aspect implies that at any steps of the process the analyst can discover a construct that he is not looking for (opportunistic approach). He must be able to represent this construct in the schema. For example, if during the analysis of the source code of the program, to refine the physical schema of a relational database, he discovers that a table is a sub-type of another, he must be able to note it in the physical schema even if it is a conceptual model concept.

DBRE tries to recover the conceptual schema of the persistent data of a legacy application. It is not exceptional that a legacy system does not have only one database but several and possibly of different paradigms. This may append because the system is old and functions have been added. These added functions may require other data and the programmer has decided to create a new database (of a different paradigm) to store them. Another origin for more than one database for an application is the integration of different applications. At the beginning there was two different applications (possibly of different companies or departments) and these applications were merged into a single application, but the databases were not merged into a single one.

For all those reasons, it is necessary that our model must be able to represent schemas with different levels of abstraction and different paradigms.

2.3. DMS-specific data structure specification

This section describes how the generic model represents data structures that can be found in legacy systems. Four different families of models will be studied: relational model (SQL), network model (CODASYL), hierarchical model (IMS) and standard file model (COBOL). For each model, the translation of the model concepts to the generic model concepts is described: how the data structure

can be specified and what are the differences between the logical and the physical schema. We will also describe how different implicit data structures and constraints can be specified.

Relational Model	Generic model
Tablespace, Storage area, DBspace	Collection
Table	Entity type
Column	Atomic single-valued attribute (.-1)
Column not null	Mandatory attribute (1-1)
Column null	Optional attribute (0-1)
Primary key	Primary identifier
Index	Access key
Unique index	Secondary identifier and access key
Foreign key	Reference group

FIGURE 4. Relational model concepts translation.

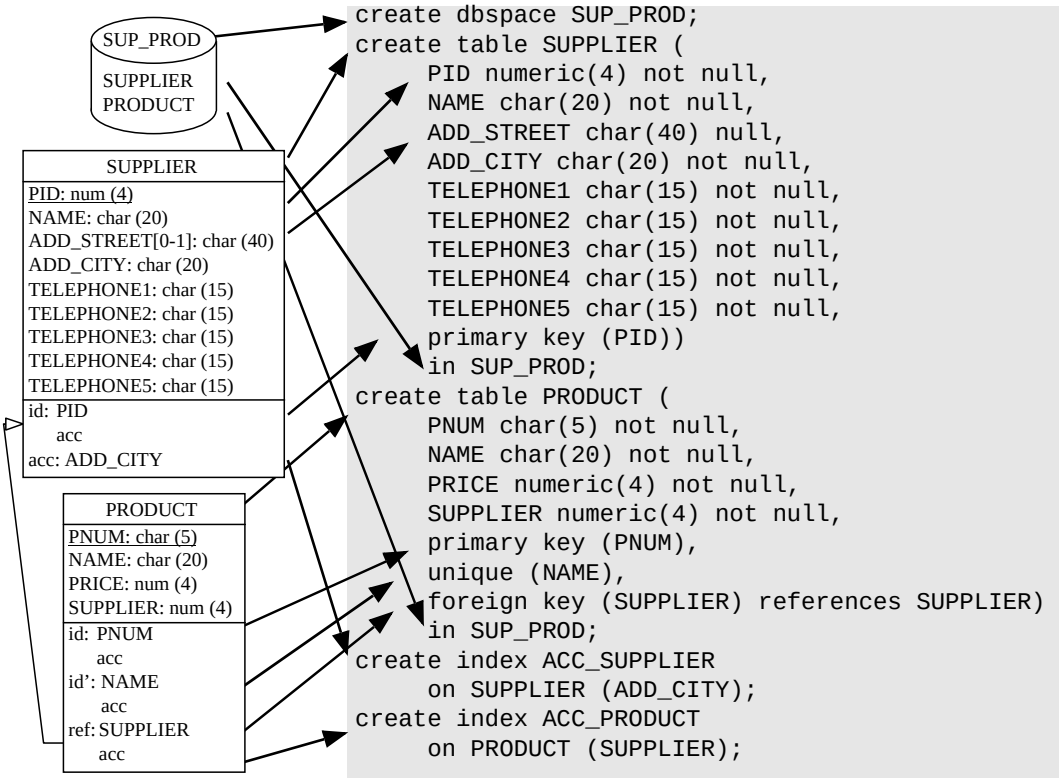


FIGURE 5. SQL physical schema with its corresponding creation SQL-DDL code.

2.3.1. The relational model

The relational model does not contain a relationship type. Only atomic single-valued attributes are accepted, they are called *columns* and they can be optional or mandatory. The main explicit constraints are the primary identifier, called *primary key*, the secondary identifier, called *unique*

index and the referential constraint, called *foreign key.* Access keys can be declared with duplicates or not and are called *index* or *unique index*. Each identifier is supported by an access key (index). An entity type is called a *table*. A collection, called *tablespace*, *storage area* or *dbspace* depending on the DMS, can contain more than one entity type.

The table of the figure 4 summarizes concepts translation between the generic model and the relational model. Figure 5 is an example of an SQL schema and of the SQL-DDL code used to create such a database.

2.3.2. The network model

Network Model	Generic model
Area	Collection
Record	Entity type
Field	Mandatory attribute (1-1)
Array (occurs I)	Multi-valued attribute (I-I) - array
Calc ... duplicates not allowed	Primary identifier and access key (contains only attributes)
Calc ...	Access key (contains only attributes)
Key is ... duplicates not allowed	Primary identifier and access key (contains the role of the owner and attributes)
Key is ...	Access key (contains the role of the owner and attributes)
Set	Relationship type
Owner	0-N role
Member	?-1 role
If more than one member	Multi-domain role

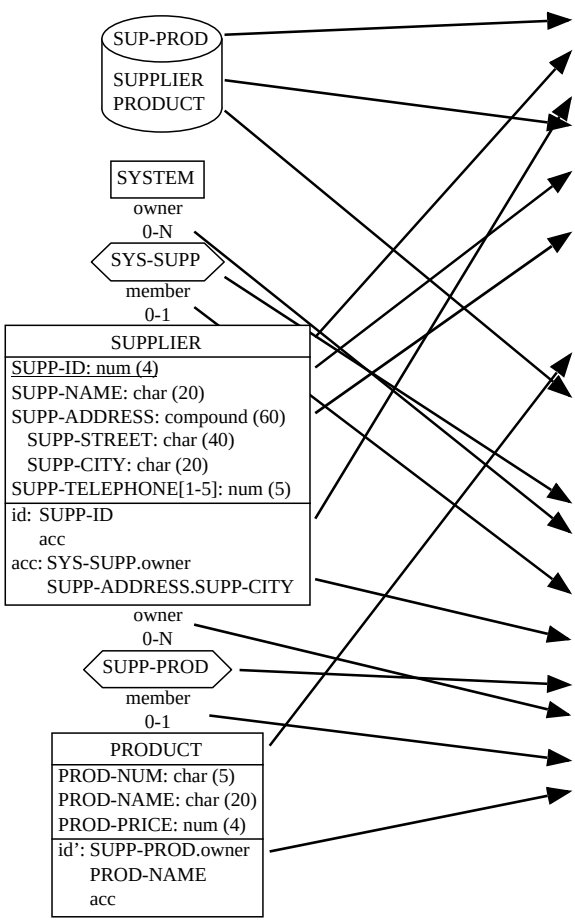
FIGURE 6. Network model concepts translation.

In the network model there is no isa hierarchy. A collection, called *area*, can contain more than one entity type, called *record*, and an entity type can be stored in more than one collection. The attributes, called *fields*, can be single-valued or multivalued (array), atomic or compound but they are mandatory. The relationship types, called *sets*, are binary, one to many, cannot be cyclic and do not contain attributes. The 0-N role is called the *owner* and the 0-1 (or 1-1) role is called the *member.*

The network model has two kinds of access key, *calc key* and *key.* The *calc key* contains only attributes. There is at most one all-attribute identifier per entity type, called *calc key ... duplicate not allowed*. The second kind of access key, called *key*, contains one member role and at least one attribute. The *key ... duplicate not allowed* is a secondary identifier.

The network model has a special entity type, called *system*, that contains no data. The system entity type is use has the owner of an entity type that has no natural owner and for which it is necessary to create a *key.*

Each entity type has an implicit technical identifier, called *DBkey*, that is the physical number of the entity type which can be used to access the entity type. The DBkey is not explicitly represented in the schema. If the DBkey needs to be represented (because it is referenced by a foreign key, for example) a technical attribute, called *DBkey*, is added and declared as an identifier.




```
AREA NAME IS SUP-PROD.
RECORD NAME IS SUPPLIER
  LOCATION MODE IS CALC USING (SUPP-ID)
    DUPLICATES ARE NOT ALLOWED
  WITHIN AREA SUP-PROD.
    02 SUPP-ID PIC 9(4).
    02 SUPP-NAME PIC X(20).
    02 SUPP-ADDRESS.
      04 SUPP-STREET PIC X(40).
      04 SUPP-CITY PIC X(20).
    02 SUPP-TELEPHONE PIC 9(5) OCCURS 5.
RECORD NAME IS PRODUCT
  LOCATION MODE IS VIA SUPP-PROD
  WITHIN AREA SUP-PROD.
    02 PROD-NUM PIC X(5).
    02 PROD-NAME PIC X(20).
    02 PROD-PRICE PIC 9(4).
SET NAME IS SYS-SUPP
  OWNER IS SYSTEM
  ORDER IS PERMANENT LAST.
  MEMBER IS SUPPLIER
  KEY IS (SUPP-CITY ASCENDING)
     DUPLICATES ARE FIRST.
SET NAME IS SUPP-PROD
  OWNER IS SUPPLIER
  ORDER IS PERMANENT LAST.
  MEMBER IS PRODUCT
  KEY IS (PROD-NAME ASCENDING)
     DUPLICATES ARE NOT ALLOWED.
```

FIGURE 7. CODASYL physical schema with its corresponding creation CODASYL-DDL code.

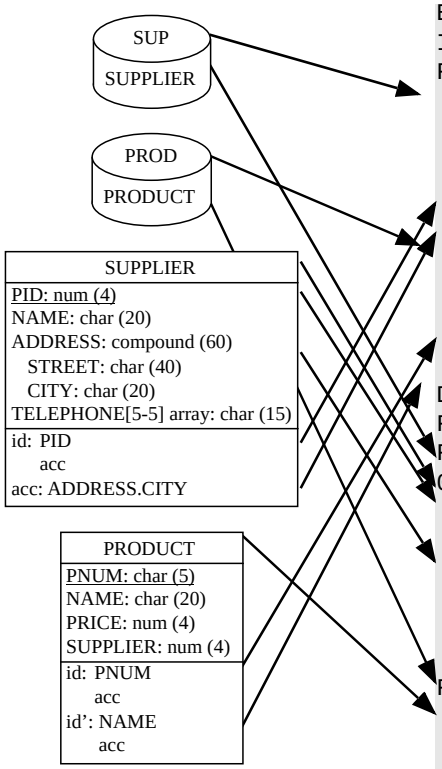
The table of figure 6 summarizes concepts translation between the generic model and the network model. Figure 7 is an example of a CODASYL physical schema and of the CODASYL DDL code used to create such a database.

2.3.3. The standard file model

The standard file model does not include the concept of relationship type nor that of foreign key. The attributes, called *fields*, are single-valued or multivalued (array), atomic or compound and are mandatory. Two fields or more can have the same physical position by the usage of the clauses *redefine* or *rename*. A collection, called *file*, can include several entity types, called *records*. An entity type can be in only one file.

Standard file Model	Generic model
File	Collection
Record type	Entity type
Field	Mandatory attribute (1-1)
Array (occurs I)	Multi-valued attribute (I-I) - array
Record key	Primary identifier and access key
Alternate record key without duplicate	Secondary identifier and access key
Alternate record key with duplicate	Access key
Redefine, rename	Group with "redef" function

FIGURE 8. Standard file model concepts translation.




```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT SUP ASSIGN TO "SUP"
     ORGANIZATION IS INDEXED
     ACCESS MODE IS DYNAMIC
     RECORD KEY IS PID
     ALTERNATE RECORD KEY IS CITY WITH DUPLICATES.
   SELECT PROD ASSIGN TO "PROD"
     ORGANIZATION IS INDEXED
     ACCESS MODE IS DYNAMIC
     RECORD KEY IS PNUM
     ALTERNATE RECORD KEY IS NAME.
DATA DIVISION.
FILE SECTION.
FD SUP.
01 SUPPLIER.
    02 PID  PIC 9(4).
    02 NAME  PIC X(20).
    02 ADDRESS.
        03 STREET  PIC X(40).
        03 CITY  PIC X(20).
    02 TELEPHONE  PIC X(15) OCCURS 5 TIMES.
FD PROD.
  01 PRODUCT.
    02 PNUM  PIC X(5).
    02 NAME  PIC X(20).
    02 PRICE  PIC 9(4).
    02 SUPPLIER  PIC 9(4).
```

FIGURE 9. COBOL physical schema example with its corresponding file declarations code.

The standard file model encompasses different types of file. In this section we only describe three of them (sequential, relative and indexed). There are other types that are not used to store permanent data but temporary elements, such as sort, merge and report files. The *sequential* file, as its name suggests, is a file that can only be accessed sequentially. So, there is no identifier or access key defined. In a *relative* file, the only means to access directly one of its entity types is through the entity type number, called *relative key.* This entity type number is the only identifier of the record. To represent a relative file, a technical attribute is added to the entity type that represents the record

number and this attribute is the primary identifier of the record and it is supported by an access key. In an *indexed* file, the user can define its own identifiers and indexes. The primary identifier, called *record key* is supported by an access key. A secondary identifier, called *alternate record key without duplicate*, is supported by an access key. Non identifying access keys are called *alternate record key with duplicate*.

The table of figure 8 summarizes concepts translation between the generic model and the standard file model. Figure 9 is an example of a COBOL physical schema with its declaration code counterpart.

2.3.4. Other constructs

As shown in the previous pages, DMS models do not have very rich set of data structures and constraints. A lot of constructs could not be expressed in the DMS model. For example, the COBOL model does not have reference constraints nor relationship type. None of the DMS model offers mechanism to indicate that there is some redundancy. It is not because the DMS model does not offer some constraints that the programmer does not need them. In such situations, the programmer implements, implicitly, these constraints in the application program.

During DBRE, if the analyst discovers through some code or data analysis, a constraint or data structure that does not belong to the DMS model, he must express it in the schema. If the constraint exists in another DMS model or in the conceptual model, the other model notation is used. For example, a reference constraint discovered during the analysis of COBOL files is noted as a reference group into the schema. Another example is an attribute of a relational database that can be decomposed into several attributes (the address can be decomposed as street, city, zip code) is represented as a compound attribute.

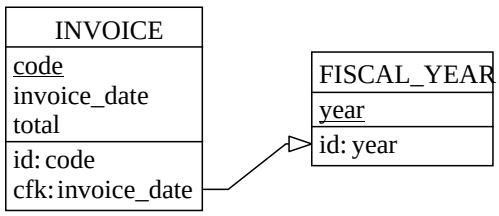


FIGURE 10. Computed reference constraint example.

It is necessary to define some techniques to represent non standard constraints, i.e. constraints that do not appear in DMS models nor in text books but that programmer use. We have identified some of them:

- *Computed referential constraint (cfk)*

  In standard foreign key, the target and the origin of the foreign key have the same value. Sometimes a function is applied to the value of the origin to obtain the value of the target. For example, in figure 10, *invoice_date* is the date of the invoice and *year* is the number of the year. The later can be extracted from the *invoice_date*. The computed foreign key is represented by an inter-group constraint where the first group is the reference group (noted *cfk*, for computed foreign key) and the second group is the reference identifier with the expression of the matching function recorded as a annotation.

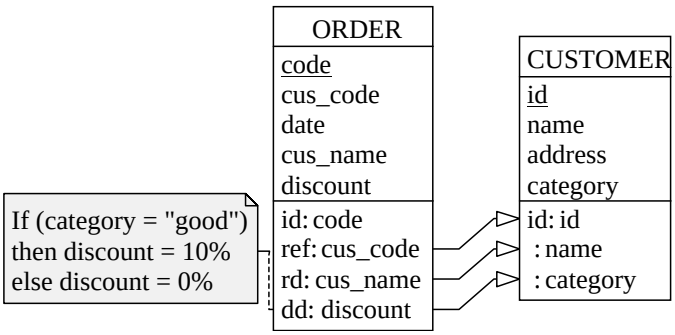


FIGURE 11. Redundancy and data dependency constraint example.

- *Redundancy constraint (rd)*

  A common optimization technique consist in copying some information from one record to another. In the schema of figure 11, *CUSTOMER.name* is copied into *ORDER.cus_name*. The redundancy constraint is represented as an inter-group constraint representing that the value of the origin group is a copy of the target group. The target group has no specific type.

- *Data dependency constraint (dd)*

  Data dependency is an inter-group constraint representing that every instance of the origin group depends on the value of the target group. In the schema of figure 11 *discount* is data dependant on *category*. The annotation gives the function to compute the value of *discount*.

- *Obsolete / unused data structure*

  During the maintenance and evolution of the database, it is possible that some attributes or entity types were created and are not used any more. It is important to mark these data structures as obsolete to prevent any further analysis and to suggest to suppress them later, during some maintenance or migration process.

- *Working data structure*

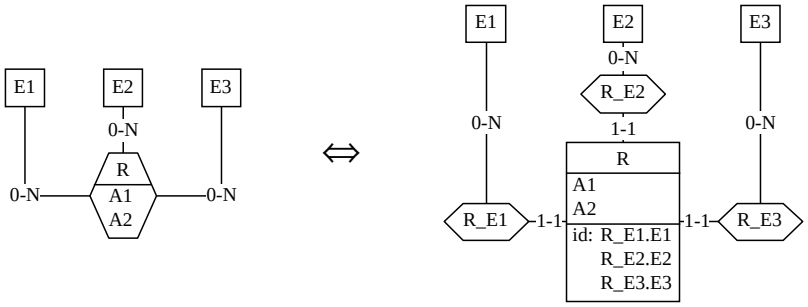
  Entity types or attributes found in the database can have no relation with the application domain, they are *working data structures*. For example, it can be stored in an entity type the login and password of the user of the application, the last customer's number used. These data structures are dependent on the way the application is implemented and not on the domain of the application. For example, if the operating system offers some access control to the application, it is not necessary to maintain an entity type with the login and password.

Other type of constraints can be found, for which no standard representation has been defined. It is up to the analyst to define and document his own notations to represent them.

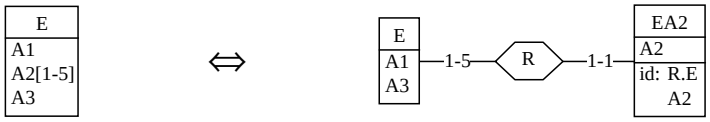
2.4. Schema transformation

Most database engineering processes can be modeled as data structure transformation. Indeed, the production of a schema can be considered as the derivation of this schema from a (possibly empty) source schema through a chain of elementary operations, called *schema transformations*. Adding a relationship type, deleting an identifier, translating names or replacing an attribute with an equivalent entity type, are all examples of basic operators through which one can carry out such engineering processes as building a conceptual schema [Batini et al.-1992][Batini et al.-1993], schema

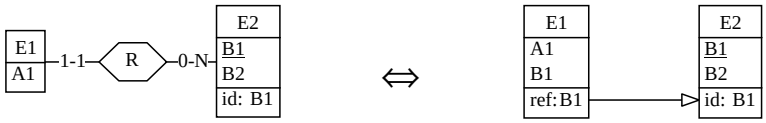
normalization [Rauh et al.-1995], DMS schema translation [Hainaut et al.-1993b][Rosenthal et al.-1988][Rosenthal et al.-1994], schema integration [Batini et al.-1992], schema equivalence [D'Atri et al.-1984][Jajodia et al.-1983][Kobayashi-1986][Lien-1982], data conversion [Navathe-1980], schema optimization [Hainaut et al.-1993a][Halpin-1995] and others [Blaha-1996][De Troyer-1993][Fahrner et al.-1995]. As it will be shown later on, they can be used to reverse engineer physical data structure as well [Bolois et al.-1994][Casanova et al.-1984][Hainaut et al.-1993a][Hainaut et al.-1993b][Hainaut et al.-1995].



a) Relationship type into an entity type.



b) Attribute into an entity type by instance representation.



c) Relationship type into a reference group.

FIGURE 12. Example of semantic-preserving transformations.

A (schema) *transformation* is most generally considered as an operator by which a source data structure *C* is replaced with a target structure *C'*. Though a general discussion of the concept of schema transformation would include techniques through which new specifications are inserted (*semantics-augmenting*) into the schema or through which existing specifications are removed from the schema (*semantics-reducing*), we will mainly concentrate on techniques that preserve the specification (*semantics-preserving*). A complete discussion about schema transformation can be found in [Hainaut et al.-1993b]. Some examples of semantic-preserving transformations are given in figure 12. The first transformation is the transformation of a relationship type into an entity type and its inverse (figure 12.a). The second transform an attribute into an entity type by instance representation, in which each instance of *A2* in each entity type is represented by an *EA2* entity type and its inverse (figure 12.b). Finally, figure 12.c shows the transformation of a relationship type into a reference group and its inverse.

```
for each isa hierarchy (isa)
        transform isa into relationship type
for each complex[1] relationship type (rt)
        transform rt into entity type
for each binary many-to-many relationship type (rt)
        transform rt into entity type
while there is some multivalued or compound attribute
        for each multivalued attribute (att)
                transform att into entity type
        for each compound attribute (att)
                disaggregate att
for each relationship type (rt)
        transform rt into reference group
                (add technical identifier if necessary)
```

1. relationship type with more than one role or with attribute.

FIGURE 13. Transformation plan to transform a conceptual schema into its logical relational equivalent.

Being functions, transformations can be composed in order to form more powerful operators. Complex transformation combinations can be built through *transformation plans* which are high level semi-procedural scripts that describe how to apply a set of transformations in order to fulfill a particular task or to meet a goal. These scripts are composed of transformations that are executed if some predicates (about properties of the schema) are satisfied. It is important to note that a transformation plan can be considered as a strategy for higher level transformation to be performed on a whole schema. For example, the transformation plan to transform a conceptual schema in its logical relational equivalent is given in figure 13.

This analysis leads to an important conclusion for the following: all engineering processes, be they related to forward or reverse engineering, can be considered as schema transformations.

CHAPTER 3 *A generic methodology for database reverse engineering*

The database reverse engineering (DBRE) methodology is introduced as the reverse of the database forward engineering. A generic DBRE methodology is presented. This method is divided into three processes: project preparation, data structure extraction and data structure conceptualization. And it produces two schemas, the logical schema and the conceptual schema. The project preparation identifies the components to be analyzed, the resources to be allotted and the planning. The data structure extraction process aims at rebuilding the logical schema of the database. This schema can be described as the database view the programmer has (or must have). The data structure conceptualization process tries to specify the semantic structure of the logical schema as a conceptual schema.

3.1. Database reverse engineering is the reverse of forward engineering

Since reverse engineering consists in recovering (among others) the conceptual schema from the operation code, it is reasonable to consider that this process is just the reverse of forward engineering [Baxter-1997].

Reversing a hierarchically decomposable process consists in inverting the order of the sub-processes, then replacing each sub-process with its inverse. For databases that have been developed according to an ideal approach, the output of forward engineering, thus the input of reverse engineering process, is of three kinds: $Code_{ddl}$ represents the DDL code, $Code_{ext}$ represents the non-DDL code (part of the structure and constraints that are not expressed in the DDL code) and $E(\Delta)$ is the part of the specification that is lost during the forward engineering. The forward engineering is composed of the following processes (left part of figure 14):

- *Conceptual design*

  Produces the *conceptual schema*, a computer-independent description of the information structures to be implemented by the database. It is divided in two sub-processes:

  - *Conceptual analysis*: translates user's requirements in formal specifications.

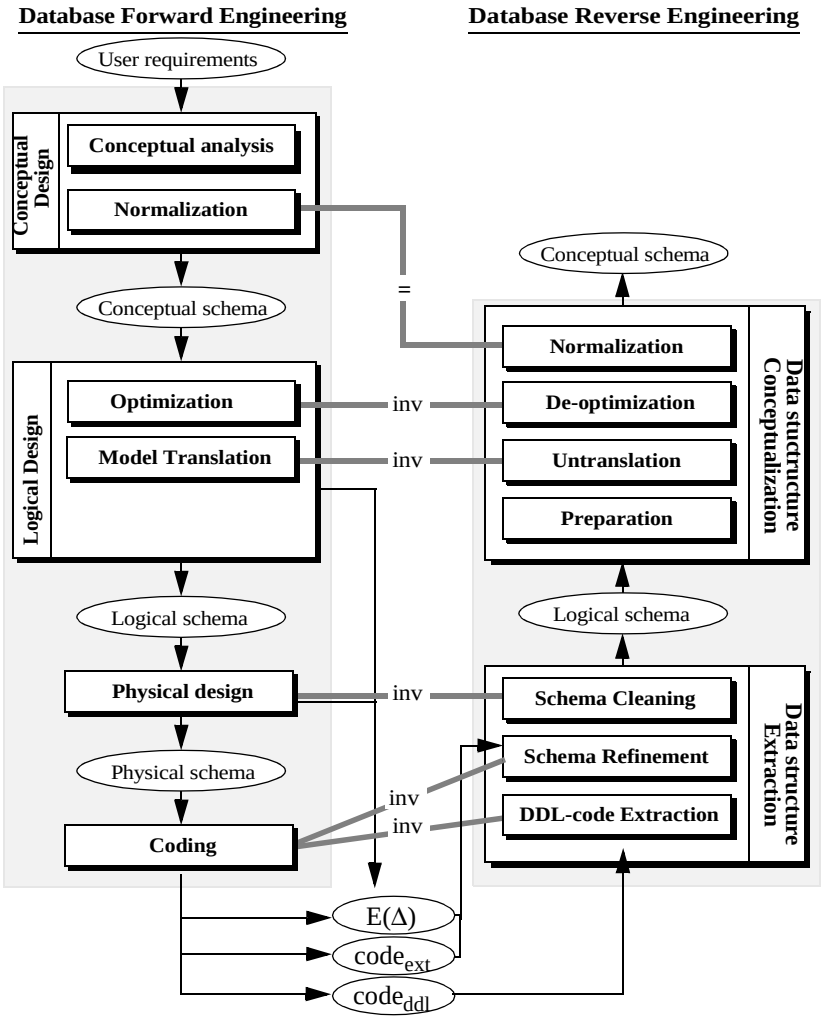


FIGURE 14. The main DBRE processes as the inverse of forward processes.

- *Normalization*: gives the conceptual schema desirable qualities such as normality, minimality, readability, clarity and compliance with corporate standards.

- *Logical design*

  Transforms the conceptual schema into a DMS-compliant optimized logical schema (called *logical schema*). This schema is expressed in the data model of the chosen DMS and it satisfies operational criteria such as space and time performance. It is divided in two sub-processes

  - Optimization: modifies the schema in order to give it better performance.
  - Model translation: converts the schema into data structures that are compliant with the model of the DMS.

  Some constructs of the conceptual schema are not transformed into the logical schema (E(Δ)). This lost of semantics happens because the DMS model is weaker than the conceptual model and some constructs are to complex (expensive) to express or the program does not know how to express them.

- *Physical design*

  Technical parameters are set and physical constructs are defined to generate the *physical schema*.

- *Coding*

  Produce an executable version of the database. Translate the DMS constructs into DMS-DDL definition (called the $code_{ddl}$) and non-DMS constructs into, e.g., procedural sections (called $code_{ext}$). Some constructs are not translated (intentionally or not) in code (E*(Δ)*).

If DBRE is the reverse of those processes, it can be sketched graphically (figure 14) in order to show the links with the forward process. The correspondence forward / reverse marked with "inv" means that each process is the inverse of the other, while the symbol "=" indicates that they are of the same nature. DBRE comprises two steps:

- *Data structure extraction*

  Recover the logical schema from the operational code ($code_{ddl}$ and $code_{ext}$). This consists in uncoding the $code_{ddl}$ (*DDL-code extraction*) and uncoding the $code_{ext}$ (*schema refinement*). Some parts of the non implemented specification *(E(Δ))* can be recovered during the schema refinement through the analysis of other information sources such as the data, the user or programer interview, etc. To obtain the logical schema, the physical design process should be undone. This is fairly simple since the forward process consist in adding technical constructs to the logical constructs; this sub-process is called *schema cleaning*.
- *Data structure conceptualization*

  Recover the *conceptual schema* from the logical one, i.e. removing and transforming the optimization called *de-optimization* (reverse of the optimization), interpreting the logical constructs in terms of their conceptual structures called *untranslation* (reverse of the model transformation) and gives the conceptual schema desirable qualities (normalization).

This approach seems correct if the application that is reengineered has been developed according to an ideal approach. But a lot of applications have been developed according to some empirical approach or have evolved. How can the suggested methodology be applied to such applications?

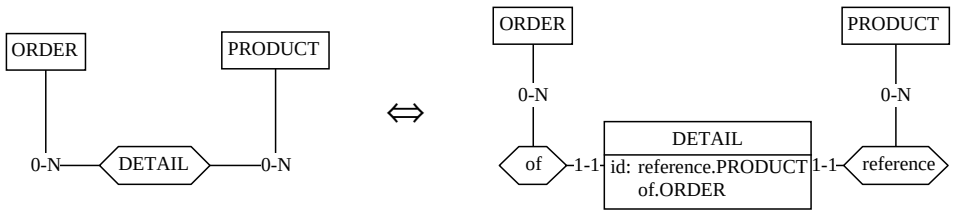


FIGURE 15. Two equivalent conceptual schema.

The purpose of the DBRE is not to recover *the* conceptual schema that was used during the conception of the database, but to recover *a* possible conceptual schema that expresses the semantics of the database. More than one conceptual schema can specify the same database. All the schemas have the same semantics, i.e. they represent the same domain. For example, the schemas of figure 15 both represent the same domain (orders, details and products) and are equivalent. To choose between one of them as the conceptual schema is a matter of corporate standards, habits, methodological standard, etc. Thus, even if the conceptual schema never existed, it is possible to recover a conceptual schema that represents the database.

A DBRE difficulty is that a part of the semantics may lie outside the system, i.e., it has not been wired in the coded part of this system (*E(Δ)*). The system (DMS, application, etc.) is not aware of the existence of these constructs. This semantics can be found elsewhere, for instance in the envi-

ronment of the application, in the documentation or in the data. Therefore, the extraction process needs to analyze other source of information than the code of the application.

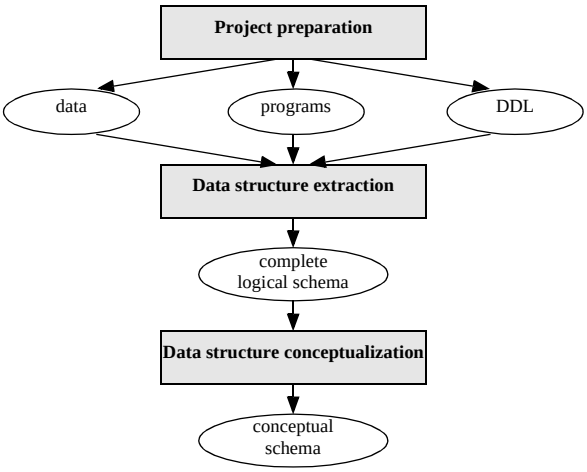


FIGURE 16. General architecture of the reference database reverse engineering methodology.

3.2. The DBRE methodology

The general architecture of the reference DBRE methodology is outlined in figure 16.

Project preparation is a preliminary step, which aims at the identification and the evaluation of the components to be analyzed, the evaluation of the resources necessary and the definition of the planning of the operations. Even if this step is not strictly a reverse engineering task but more a managerial one, experience teaches us that managing the whole project and identifying the relevant information sources is not an easy task and is crucial for the correct development of the project. The project preparation is composed of the following processes:

- *Identification of the relevant components and of their quality*

  The files, programs, screens, reports, forms, data dictionaries, repositories, program sources, data, and documentation are identified and evaluated, not all the components bring the same quality and quantity of information. For example, DDL gives very precise information and is cheap to analyze, procedural source code gives also precise information but its analysis is quite expensive. Documentation, if up-to-date and carefully written, can be very useful and cheap to analyze (and even make the reverse engineering process useless!). When the documentation is obsolete and/or not structured, its analysis can take times and, even worse, lead to false assumptions.
- *Architecture recovery*

  It consists in drawing the main procedural and data components of the system and their relationships.
- *Project definition*

  It defines precisely the part of the application that will be analyzed and what are the expected results. Usual only a part of the applications will be reverse engineered. It is important to define precisely the border of the analyzed applications, because all the applications deeply interact and it can be difficult to know if, for example, an entity type belongs to the analyzed application or

not. It is also important to explain to the customer which results he can expect to avoid surprise/ conflicts during or after the project. The nature of the result must be specified (the expected result will be a conceptual schema, a logical schema or a new database) and its weakness and strength explained (for example, only the structure of the entity types will be discover).

- *Resource identification*

  Evaluates the resources needed in terms of skill, work force, calendar, machine, tools and budget.

- *Operational planning*

  For each step of the project a completion date and a budget is given. It is important to schedule some meeting where the state of the project is presented, the difficulties met are explained and the planning can be adjusted.

The *data structure extraction* process aims at rebuilding a complete logical schema in which all the explicit and implicit structures and properties are documented. The main difficulty is that many constructs and properties are implicit, i.e. they are not explicitly declared, but they are implemented in procedural sections of the programs. Recovering these structures uses all the available information sources to extract explicit and implicit constructs.

The *data structure conceptualization* process tries to specify the semantic structures of this logical schema as a conceptual schema. While some constructs are fairly easy to interpret (e.g., a standard foreign key generally is the implementation of a one-to-many relationship type), others use tricky implementation and optimization techniques.

3.3. Data structure extraction

This phase consists in recovering the complete logical schema, including all the implicit and explicit structures and constraints. This schema can be described as the database view the programmer has (or must have) to develop new applications and to maintain them correctly. So this schema contains a description of all the collections, the entity types and attributes of the database, with their physical names and all the constraints, implemented or not, that the data must verify.

True database systems generally supply, in some readable and processable form, a description of this schema (data dictionary contents, DDL text, etc.). Though essential information may be missing from this schema, the latter is a rich starting point that can be refined through further analysis of the other components of the application (views, sub-schemas, screen and report layouts, procedures, fragments of documentation, database content, program execution, etc.). The problem is much more complex for standard files, for which no computerized description of their structure exists in most cases. The analysis of each source program provides a partial view of the collection and entity type structures only. For most real-world applications this analysis must go well beyond the mere detection of the entity type structures declared in programs.

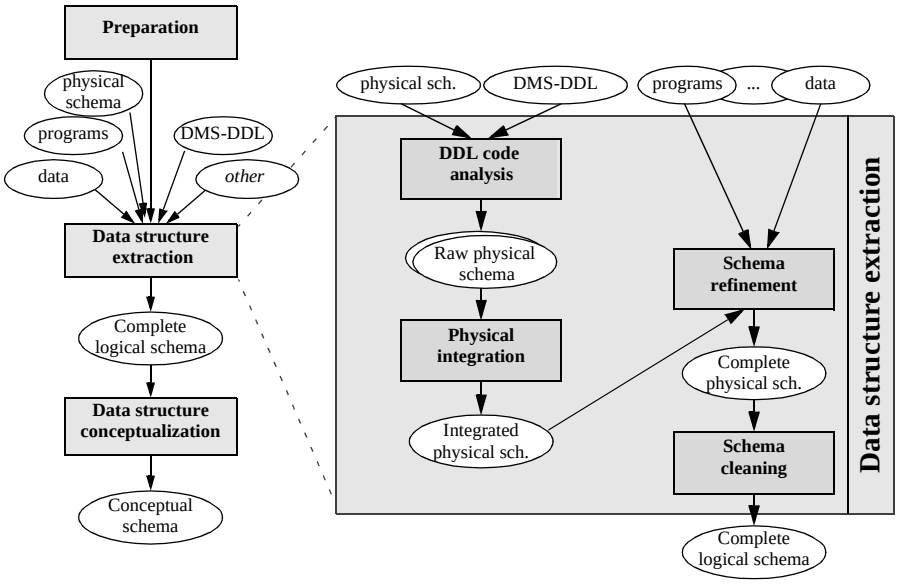


FIGURE 17. General architecture of data structure extraction phase.

The main processes of data structure extraction are shown in figure 17.

3.3.1. DDL code analysis

This rather straightforward process consists in analyzing the data structure declaration statements (in specific DDL) included in the schema creation scripts and/or application program declarations (see annex for a detailed description of the DB-MAIN DDL code extractors). It produces a physical schema, called *raw physical schema*. Extracting physical specifications from the system data dictionary, such as DMS system catalog, is of the same nature as DDL analysis.

3.3.2. Physical integration

When more than one DDL source has been processed, the analyst is provided with several extracted schemas. This can occur when the DMS does not provide a unique data dictionary. For example each COBOL program declares the files it uses and thus each COBOL program declares a part of the files used by the application. Other DMS, as CODASYL declares a global schema that gives the list of the entity type and the identifier and access keys. The sub-schemas do not define the identifiers and access keys but can declare different (more precise) data structures of the entity type. With such DMS, the global schema and the sub-schema must be integrated. A last situation with several physical schema is when the application uses several different databases of possibly different DMS. The analysis of each database produces a physical schema and all those schemas need to be integrated.

3.3.3. Schema refinement

The *schema refinement* process is a complex task through which various information sources are searched for evidence of implicit or lost constructs. The explicit physical schema obtained so far is

enriched with these constructs, thus leading to the *complete physical schema*. The complexity of the process mainly lies in the variety and in the complexity of the information sources. Indeed the implicit constraints are hidden, among others, in procedural sections in the application programs, JCL scripts, GUI procedures, screens, forms and reports, triggers and stored procedures. To perform reverse engineering of one application, usually more than one of those potential sources of information need to be analyzed and for each one there exists more than one way to express a constraint. For example, in a COBOL program there are at least six different ways to validate a reference constraint.

In addition, the non encoded part of the system must be analyzed as well because it can provide evidence for lost constructs. This part includes file contents (the data), existing documentation, experimentation (execution of the program), user and programmer interviews as well as the environment behavior. Environment behavior are the constraints that are enforced by the environment of the application. For example, the list of the customers is provided by another application that verifies all the constraints and thus the current application does not validate that the customer number is unique (an identifier). So it is impossible to discover the identifier of the customer by the analysis of the current application.

3.3.4. Schema cleaning

This process transforms the *complete physical schema* into the *complete logical schema* by removing or transforming all the physical constructs into logical ones. All the physical constructs can be discarded at this point because they do not provide any information about the database logical structure. They were useful for technical reasons such as optimizing the performance of the database or implement some access mechanisms, due to DMS limitations. The main transformations that can be applied are:

- Removing access keys.
- Removing collections.

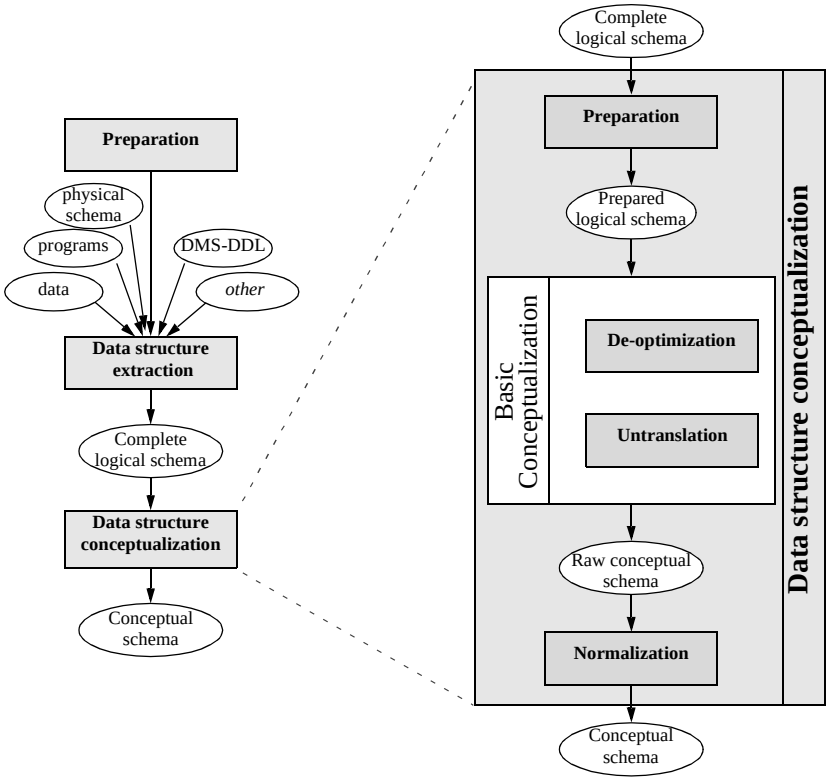


FIGURE 18. General architecture of the data structure conceptualization phase.

3.4. Data structure conceptualization

This second major phase addresses the conceptual interpretation of the logical schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimizations and DMS-dependent constructs and in interpreting them. The conceptualization phase comprises three main processes, namely *preparation*, *basic conceptualization* and *conceptual normalization* (figure 18).

- *Preparation*

  The *complete logical schema* obtained so far may include constructs that must be identified and discarded or transformed because they convey no semantics. There are two main kinds of such constructs. The *dead data structures* are obsolete structures that have been carefully left in the database by the successive programmers. The *technical data structures* have been introduced as internal constructs of the programs and are not intended to model the application domain (program counter, name of the last user, etc.).

- *Basic conceptualization*

  The main objective of this process is to extract all the relevant semantic concepts underlying the *prepared logical schema*. Two different problems, requiring different reasoning and methods, have to be solved: schema untranslation and schema de-optimization.

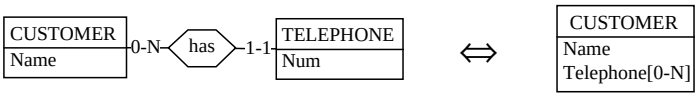
  The logical schema is the technical translation of conceptual constructs. Through *schema untranslation*, the analyst identifies the traces of such translations and replaces them with their

equivalent conceptual constructs. Though each data model can be assigned its own set of translating (and therefore of untranslating) rules, two facts are worth mentioning. First, several data models can share an important subset of translating rules (e.g. COBOL files and CODASYL structures both have multivalued attributes but not optional attributes). Secondly, translation rules considered specific to a data model are often simulated in other data models (e.g. foreign keys in IMS and CODASYL databases). Hence the importance of generic approaches and tools.

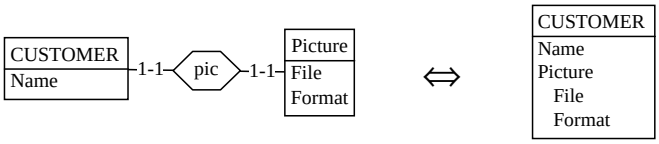
The logical schema is searched for traces of constructs designed for optimization purpose, this activity is called *schema de-optimization*. Four main families of optimization techniques should be considered: discarding constructs, denormalization, structural redundancy and restructuring.

- *Conceptual normalization*

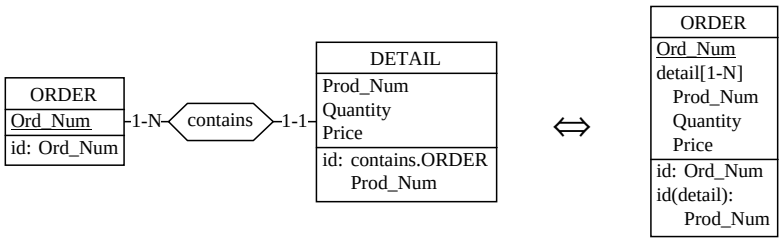
  This process restructures the basic conceptual schema in order to give it the desired qualities one expects from any final conceptual schema, such as expressiveness, simplicity, minimality, readability, genericity, extensibility. For instance, some entity types are replaced with relationship types or with attributes, is-a hierarchies are made explicit, names are standardized, etc.



a) Untranslation transformation.



b) De-optimization transformation.



c) Normalization transformation.

FIGURE 19. Example of entity type transformed into attribute used in different process.

The transformations used in each step will not be detailed, but section 3.4.4 gives the list of all the transformations used in reverse-engineering with their forward counterpart and in which process they are usually used. Several of those transformations are not used only in one process and need to be described more than once. For example, the transformation of an entity type into an attribute can be used in the untranslation, the de-optimization and in the normalization processes, depending of the DMS and the schema:

- *Untranslation*

  A relational database does not support multivalued attribute. One solution to store a multivalued attribute in a relational database is to transform it into an entity type. During the untranslation process if the analyst find an entity type that contains only one attribute and has only one role of cardinality 1-1, this entity type can be untranslated as an attribute (see figure 19.a).

- *De-optimization*

  If an entity type is accessed quite often and it contains a huge attribute that is rarely used, a common optimization is to transform this attribute into an entity type. For example, in figure 19.b, if the `CUSTOMER` entity type contains the picture of the customer and this entity type is accessed each time the customer orders a product to have its name, but the picture is used only once a year. During the de-optimization an entity type that is connected by a one-to-one relationship type can be transformed into an attribute.

- *Normalization*

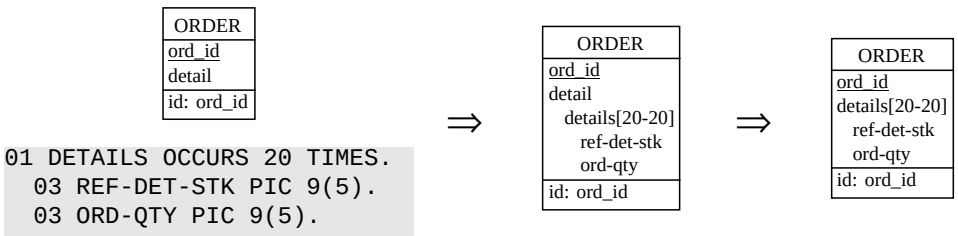
  An entity type, that plays only a 1-1 role and is dependent on the entity type to which it is connected, can be transformed into an attribute during the normalization of the schema. For example, in figure 19.c, `DETAIL` plays a 1-1 role with `ORDER`, it can be transformed as a multivalued attribute of `ORDER`.

3.4.1. Preparation

This phase prepares the schema in such a way that it only contains structures and constraints that are necessary to understand the semantics of the schema. They are two kinds of constructs that need to be discarded: *dead data structures* and *technical data structures*.

In addition, this phase carries out cosmetic changes such as naming improvement and physical construct removing. DMS impose restrictions about the name of the objects (not all the characters are accepted, the length of the names is limited, names are case sensitive or not). A database has been developed by several programmers and has been maintained, so the naming convention are not always the same: synonyms have been used (in some entity type the customer term has been used, in other the term buyer is used, etc.), more than on language has been used (mixture of English, French and Dutch word). Through screen and report analysis meaningful names can be found. For all those reasons it is necessary to rename the objects with meaningful name to have a more readable schema.

The logical schema still contains physical constructs that are useless in the conceptual schema such as indexes files. These constructs can be removed.



a) Raw schema and variable declaration. b) Refined schema. c) Prepared schema.

FIGURE 20. Example of unnecessary attributes decomposition.

Some awkward parts of the schema can be also restructured. For example, during data structure extraction, the structure of some attributes is refined. During this phase, unnecessary attribute decomposition can be created. Figure 20.a shows the physical schema and the data structure declaration used to refine the attribute *Detail*. The result (figure 20.b) gives the attribute *Detail* that is

composed of only one attribute (*Details*). This unnecessary decomposition can be suppressed (figure 20.c).

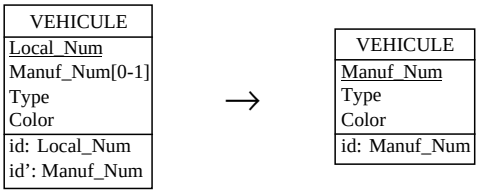


FIGURE 21. The *local_num* attribute is not used any more and can to be removed.

3.4.1.1. Dead data structures

The *dead data structures* are obsolete data structures, which have been carefully left in the database by the successive programmers. These data structures typically appear during the evolution of the applications. For example, a car seller gives to each of the vehicles he sells an identification number and the car manufacturer also gives (another) identification number to the car. The two numbers are needed, because when the order is created the manufacturer number is not yet known. So the car description entity type contains two identifying attributes, see figure 21, one for the number generated by the seller (*Local_Num*) and one for the number given by the manufacturer (*Manuf_Num*). After some years, the car seller has a direct access to the manufacturer information system and immediately knows the car number given by the manufacturer. So he decides to use *Manuf_Num* as the only identifier, so the attribute *Local_Num* becomes redundant, but the programmer does not remove the attribute from the database, he just discard its uses. During the data structure extraction phase we have noticed that the field is not used anymore and it can be removed.

Another origin of dead data structure is that the programmer anticipates a possible evolution of the application. For example, the programmer notices that the supplier has an E-mail address, while the customer has none. He adds such a attribute to the customer entity type to anticipate a possible evolution. Years later, this attribute was never used and it can be removed.

Programmers are reluctant to remove such data structures during classical maintenance because they need to be sure that the data structures are not used any more and usually they only master a part of the application. Another reason is that to remove a data structure (record or field), most of the DMS (see [Hick-2001]) require the creation of a new database and to download the old data into the new database. This is a non trivial operation: all the applications that use the data need to be shutdown, it takes a lot of time (to download the data) and space (to create the temporary database).

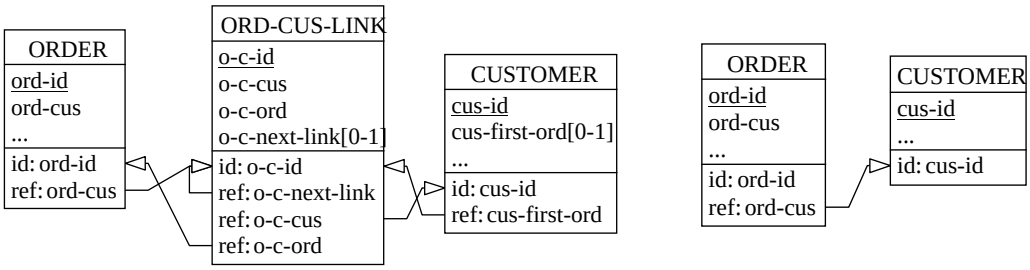
For very big databases, with a lot of data, it cannot be done in one night. It has to be done during a holiday where the enterprise is closed to perform this download.

Several hints can help to identify dead data structures. The main one is that they are not referenced (read or write) by any programs or only by dead sections of programs. We can also notice that they have no instances or their instances have not been updated for a long time.

3.4.1.2. Technical data structures

Technical data structures have been introduced as internal constructs by the programmer for technical reasons and are not intended to model the application domain. An example of such structures are

various program counters (number of records in a file, number of details in an order, etc.), name of the last user, copy of screen layouts, the list of the messages text, program save points, etc.



a) The physical schema. b) The logical schema.

FIGURE 22. A physical schema with its logical counterpart.

Those physical constructs need to be transformed into their logical counterpart (or removed). For example, access path can be explicitly implemented in the database (figure 22.a) and can be transformed into a single foreign key in the logical schema (figure 22.b).

3.4.2. Basic conceptualization

This process concentrates on extracting a first cut conceptual schema without worrying about esthetical aspects of the result. Two kinds of reasoning have been identified, namely *untranslation* and *de-optimization*. Though distinguishing between them may be arbitrary in some situations (some transformation pertain to both).

The reader will probably be surprised that the processing of some popular constructs will generally be ignored in this section. In fact, these problems have been discarded, except when they appear naturally, since they are common to all DMS and there is no general agreement on whether they must be recovered or not. Therefore they will be addressed in the conceptual normalization process. As an example, we could interpret an identifying foreign key, that is, a unique key that also is a foreign key, as the trace of a subtype/super-type relation. We prefer to transform it as a mere one-to-one relationship type, which in turn will be interpreted, in the normalization phase, either as a pure one-to-one relationship type or as entity type fragmentation or as a subtype/super-type relation.

3.4.2.1. Schema untranslation

At first glance, this process seems to be the most dependent on the DMS model. Indeed a relational schema, a CODASYL schema and a standard file schema, though they express the same conceptual schema, are made up of quite different data structures. They have been produced through different transformation plans and therefore should require different reverse untranslation rules as well. However, it can be shown that these forward plans use a limited set of common primitive transformations [Hick-2000]. A set of about 30 elementary operations has proved sufficient to define logical design strategies for all past and current DMS, from COBOL standard files to OO-DBMS ([Hainaut et al.-1996a]). Since reverse engineering is basically the inverse of forward engineering, a toolbox comprising the inverse of these forward transformations would count no more than one or two dozens of operators.

3.4.2.2. Schema de-optimization

Both conceptual and logical optimization processes will be considered as a whole since they use the same set of transformations, though possibly through different strategies. Let us recall that we have to find traces of four major families of optimization techniques based on schema transformations, namely discarding constructs, structural redundancy, unnormalization and restructuration. They must be precisely understood in order to reverse their effect. In particular, some of them are more specifically fitted for some DMS than for others.

A. Discarding constructs

This optimization resorts to the part of the specification that is lost during the forward engineering and should be addressed in the data structure extraction phase through the analysis of the data, the environmental properties and the domain knowledge.

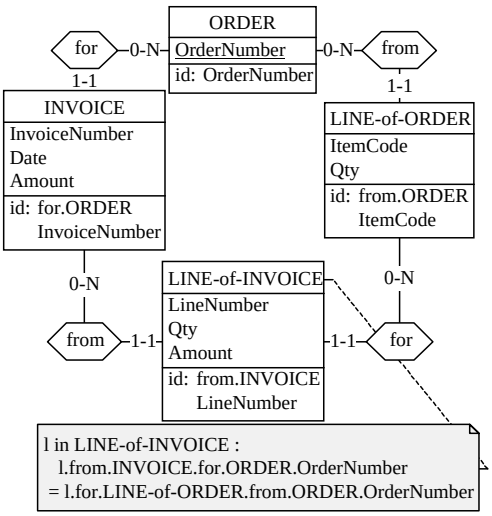


FIGURE 23. Example of expression (*l.from.INVOICE.for.ORDER.OrderNumber = l.for.LINE-of-ORDER.from.ORDER.OrderNumber*) between two constructs that do not express redundancy.

B. Structural redundancy

The main problem is to detect the redundancy constraint that states the equivalence or the derivability of the redundant constructs. The expression of such constraint is of the form $C = f(C_1,\ldots,C_n)$, where C is the designation of the redundant construct. Note that expressions such as $f_1(C_{i_1},\ldots,C_{i_n}) = f_2(C_{j_1},\ldots,C_{j_n})$ generally do not express redundancy, but rather a pure integrity constraint, in which case no construct can be removed. For example in figure 23, the schema can be interpreted as follow. There can be several invoices for an order and an invoice is associated to only one order. Each line of an invoice (*LINE-of-INVOICE*) corresponds to a (part) of the line of an order (*LINE-of-ORDER*). There exist a cycle into the schema and we can wonder if there is some redundancy in this cycle (a relationship type can be suppressed). But in this example, each relationship type is necessary because the constraint "*l.from.INVOICE.for.ORDER.OrderNumber = l.for.LINE-of-ORDER.from.ORDER.OrderNumber*" express the fact that for each line of an invoice, the number of the order (**OrderNumber**) associated with the invoice is the same as the order number of the line of the order to which the line of invoice is associated with. This is not a redun-

dancy, we could not suppress a relationship type, but an integrity constraint that need to be expressed..

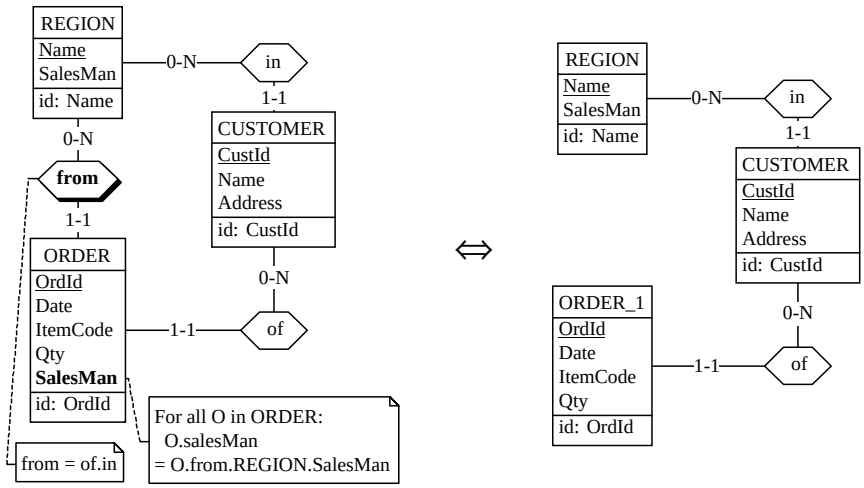


FIGURE 24. Example of structural redundancy elimination.

Figure 24 depicts the elimination of a composed relationship type and of a duplicate attribute. The attribute *ORDER.SalesMan* has been recognized during the data structure extraction phase as duplicate attribute that has the same value as *REGION.SalesMan* connected through the *from* relationship type. The relationship type *from* was recognized as the composition of *of* and *in*. So those two constructs can be removed.

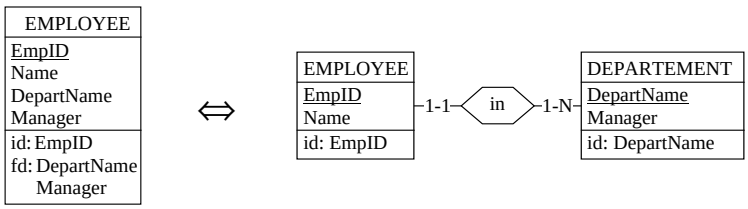


FIGURE 25. Normalization redundancy elimination.

C. Normalization redundancy

An unnormalized structure is detected in entity type *B* by the fact that the determinant of a functional dependency is not an identifier of *B*. Normalization consists of splitting the entity type by aggregating the components of the dependency as illustrated in figure 25. Note that the relationship type should be one-to-many (transforming an attribute into an entity type by value representation) and not one-to-one, otherwise, there would be no redundancy.

D. Restructuration.

We will discus the reversing of some of the more specific restructuration techniques:

- *Vertical partitioning optimization* (split)

  Entity types *E1* and *E2* are linked by a one-to-one relationship type and represent complementary properties (attributes or roles) of the same entity type.

- *Vertical merging optimization* (merge)

  Entity type *E* includes properties related to several entity types.
- *Horizontal portioning optimization* (HorPart)

  Entity types *E1* and *E2* have the same properties and represent the same kind of entities.
- *Horizontal merging optimization* (AscInher)

  Entity type *E* has unclear semantics that seem to encompass two similar but distinct entity categories.

These techniques introduce no redundancy, so it is optional to reverse them at this level. For instance, similar reasoning will be found in the normalization phase.

3.4.3. Conceptual normalization

Let us first observe that what we call *normalization* generally does not encompass the relational interpretation of the term. Indeed, relational normalization aims at removing redundancy anomalies, therefore resorting to de-optimization reasoning.

The goal of the normalization transformations is to improve, if necessary, the expressiveness, the simplicity, the readability and the extensibility of the conceptual schema. In particular, it tries to make higher level semantic constructs (such as is-a relations) explicit. Whether such expressions are desirable is a matter of methodological standard, of local culture and of personal taste. For instance, a design methodology that is based on a binary, functional ER model (e.g. the Bachman's model of the seventies) will accept most of the conceptual schema obtained so far. More powerful models will require the expression of, e.g. is-a relations or N-ary relationship types when relevant. In addition, the final conceptual schema is supposed to be as readable and concise as possible, though these properties basically are subjective.

Some constructs that need to be looked for and transformed:

- *Relationship entity type* (Et-Rt)

  By this term, we mean an entity type whose aim obviously is to relate two or more entity types. It will be transformed into a many-to-many and N-ary relationship type.
- *Attribute entity type* (Et-Att)

  Such entity type has a small number of attributes only and is linked to one other entity type *A* through a [1-j] role. All its attributes participate to its identifier.
- *one-to-one relationship type* (Merge)

  It may express the connection between fragments *B1* and *B2* of a unique entity type *B* (vertical partitioning).
- *One or several one-to-one relationship types* (Rt-Is-a)

  If the relationship types show a common entity type *A*, they may express a specialization relation in which *A* is the supertype.
- *Long entity type* (Split)

  An entity type that comprises too many attributes and roles can suggest a decomposition into semantically homogeneous fragments linked by one-to-one relationship type.

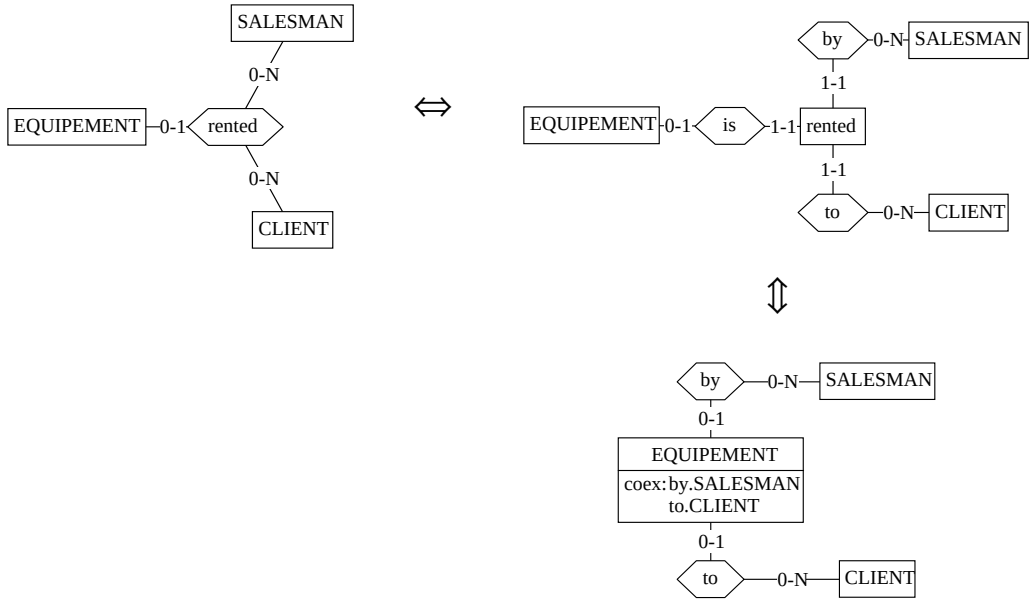


FIGURE 26. Decomposition of a N-ary relationship type through its [i-1] role.

- *N-ary relationship type with a [i-1] role* (Rt-Et + Merge)
  It can be transformed into binary many-to-many relationship types (see figure 26).
- *Entity type with common attributes and roles* (Integration(common supertype))
  If two entity types have common attributes and roles, they can be made the sub-types of a common supertype that inherits the common characteristics.

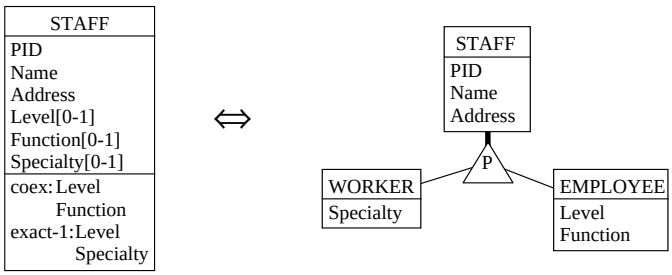


FIGURE 27. Defining subtype from coexistence subsets of optional attributes and roles.

- *Groups of coexistent attributes and roles* (Att-Et + Rt-Is-a)
  Each coexistence group can be extracted as a subtype of the parent entity type (figure 27).

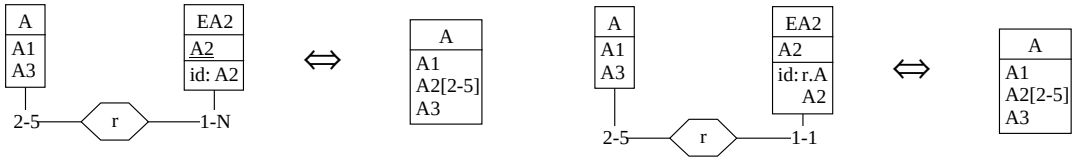
An in-depth analysis of is-a relations implementation can be found in [Hainaut et al.-1996c].

3.4.4. The data structure conceptualization transformations

In this section, the most important transformations used during the data structure conceptualization process are presented. It presents, for each transformation, the step in which it can be used (U: untranslation, D: de-optimization, N: normalization) and which forward transformation can lead to the source construct and why the developer needs this transformation. A complete description of

each transformation, with a detailed description of the pre and post conditions, can be found in [Hick-2000].

3.4.4.1. Entity type to attribute (DN)



An entity type, that plays only one role can be transformed into an attribute.

Forward transformation. There are two transformations that can lead to this structure: transformation of an attribute into an entity type by representing the possible duplicate attribute instances (by *instance*) or by representing the distinct attributes value (by *value*).

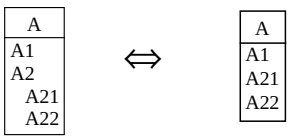
Those transformations are used during the normalization (N) to eliminate multivalued or compound attributes because the corporate standard does not allow them. They are also used during the translation (U) and optimization (D) to eliminate some constructs.

3.4.4.2. Attribute aggregation (UN)

Aggregating several attributes, those contribute to the same domain concept, as a compound attribute.

Forward transformation. Disaggregating a compound attribute. Such transformation is necessary to comply with some DMS that does not support compound attributes, as relational DMS.

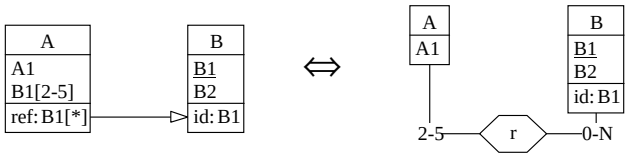
3.4.4.3. Attribute disaggregation (UN)



Disaggregating a compound attribute into its components, if the components are not of the same concept domain.

Forward transformation. Some DMS require, as COBOL, that identifiers or access keys contain only one attribute (U). So if an identifier or an access key is composed of more than one attribute, the programmer aggregate them into a technical attribute.

3.4.4.4. Foreign key into relationship type (U)



Transforming a foreign key into its relationship type counterpart. In this section, we only present the standard foreign key, according to which a set of attributes in an entity type is used to designate elements in another entity type. A careful analysis of existing databases puts into light a surprisingly large variety of non standard forms of foreign keys. Most of them are quite correct and perfectly fitted to the requirements the developer has in mind. However, their conceptual interpretation can prove much more difficult to formalize than the standard form. A complete discussion to the concept of foreign key can be found in [Hainaut-1997a].

Forward transformation. The relationship types were transformed into foreign keys because most of the DMS does not support relationship types.

3.4.4.5. List of attributes into a multivalued attribute (U)

A series of single-valued attributes, of same type and length, are transformed into a multivalued attribute.

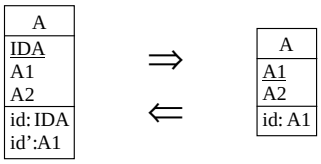
Forward transformation. Replacing a multivalued attribute with a series of single-valued attributes that represent its instance. This transformation can be used if the DMS does not support multivalued attributes, as relational DMS.

3.4.4.6. Single attribute into a multivalued attribute (U)

A single attribute is transformed into a multivalued one, the length of the original attribute must be a multiple of the length of the target one.

Forward transformation. Replacing a multivalued attribute by a single-valued attribute that represents the concatenation of its instances. This transformation can be used if the DMS does not support multivalued attributes, as relational DMS.

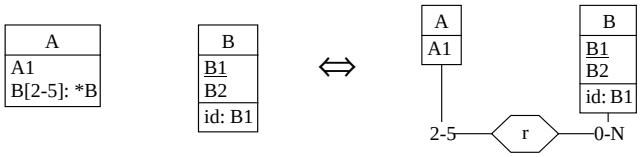
3.4.4.7. Remove technical identifier (UD)



Remove the technical identifier of an entity type if this attribute is not a concept of the domain.

Forward transformation. A semantic-less attribute is added and made primary identifier of the entity type. This can be necessary because an entity type does not have an identifier and the DMS requires that each entity type has an identifier (U) or there is an identifier but its type is too long or incompatible to use it as an identifier, e.g. as the target of a foreign key (D).

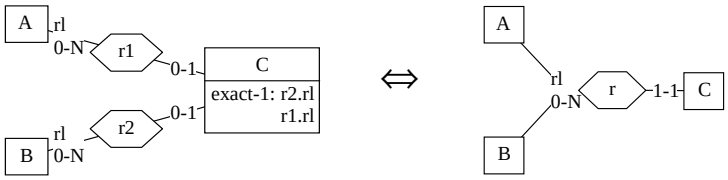
3.4.4.8. Object attribute into a relationship type (U)



Transforming an object attribute (reference or pointer) into a relationship type.

Forward transformation. Transforming a binary relationship type into an object attribute (reference or pointer). This can be useful if the target DMS is object oriented.

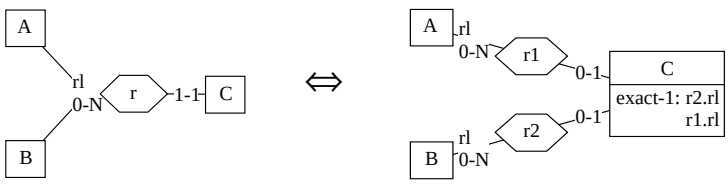
3.4.4.9. Relationship type into multi-domain role (N)



Transforming relationship types (whose roles participate in exactly one constraint) into the corresponding relationship type with a multi-domain role.

Forward transformation. Transforming a multi-domain role into the corresponding relationship types.

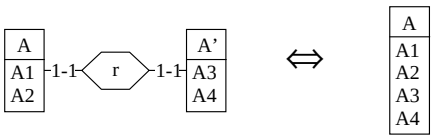
3.4.4.10. Multi-domain role into relationship type (UN)



Transforming a multi-domain role into the corresponding relationship types.

Forward transformation. Some DMS can represent multi-domain role, as the CODASYL multi-member relationships.

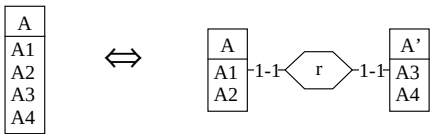
3.4.4.11. Merge entity types (DN)



Merge two entity types connected by a one-to-one relationship type into one entity type.

Forward transformation. Splitting an entity type into two entity types connected by a one-to-one relationship type. The entity type can be split because it has too many attributes and the schema is more readable with two entity types (N). It can also be split to optimize the access time (D), if some of the attributes are very often accessed and the other not very often. For example, the *product* record contains the name of the product (*name*) and its picture (*picture*); *name* is used each time product is accessed but *picture* is only used once per year to print the new product catalogue. Then *product* is split into two parts, one with the attributes that are accessed very often and the other with the remainder. Like that the record that is access very often is smaller and can be quickly loaded and once a year it takes more time to retrieve the picture of the product.

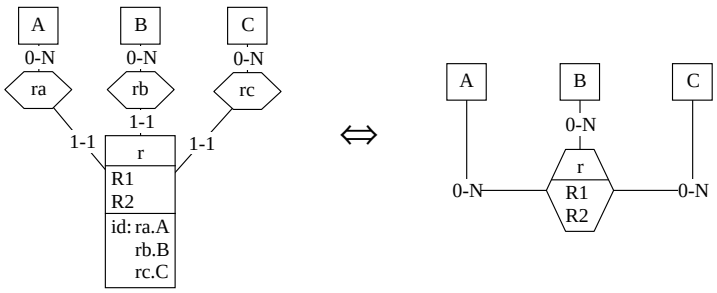
3.4.4.12. Split an entity type (DN)



Splitting an entity type that contains attributes of different application domain concepts into two entity types connected by a one-to-one relationship type.

Forward transformation. Merge two entity types connected by a one-to-one relationship type into one entity type. The two entity types represent different application domain concepts but are merged for optimization (D) purpose, if the program always need both concepts together.

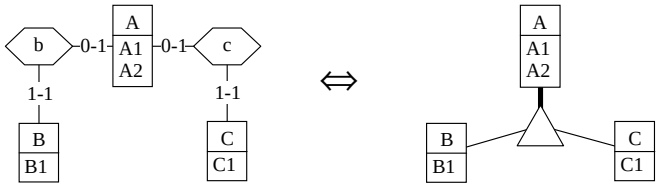
3.4.4.13. Entity type into relationship type (N)



Transforming an entity type, which seems to be the representation of a relationship type, into a relationship type.

Forward transformation. Transforming a relationship type into an entity type. DMS does not support relationship types with attributes and N-ary relationship types, they need to be transformed into entity types.

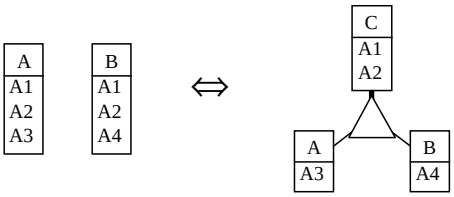
3.4.4.14. Relationship type into is-a (N)



One-to-one relationship types with a common entity type are transformed into an is-a relation.

Forward transformation. Materializing an is-a relationship into relationship types. If the DMS does not support is-a relations, they must be transformed. One of the possible transformations is to materialize them by relationship types.

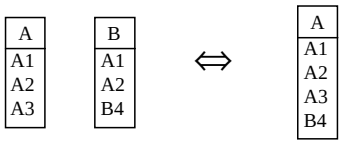
3.4.4.15. Common supertype (DN)



Transforming two entity types with common attributes or roles (attributes and roles representing same concept) by the creation of a common supertype.

Forward transformation. The attributes and roles of the supertype of an is-a relation are copied into each sub-type. If the DMS does not support is-a relations, they must be transformed. One of the possible transformation is to copy all the attributes and roles of the supertype into its sub-types.

3.4.4.16. Integration (D)



Two entity types that represent the same application domain concept are merged into an entity type.

Forward transformation. An entity type is duplicated, both entity types represent the same application domain concept. To reduce the size of a file, we can decide to horizontally partitioning it, i.e. duplicate the structure and to store one instance in the first one and the other one in the other. For example, to reduce the size of the file containing the orders, we can decide to store in one file the current orders[1] (stored on a fast disk) and the other ones that contain the archived orders (stored on another disk or on a CD-Rom or on a tape).

3.5. Example

This section presents a small DBRE example. The only source of information available is a fragment of the COBOL source code of the application (file and record declarations and some procedural code). Figure 28 sketches the DBRE process with the different products used and produced.

The first step analyzes the DDL code, i.e. translates the file declarations (the `file-control` of the `input-output section`) and the record declarations (the `file section`) to produce the physical schema. The `select` clauses are translated into collections, the `record key`'s into identifiers and the `alternate record key with duplicate` into an access key. The 01 level variables of the `file section` are translated into entity types with their respective sub-variables as attributes.

This schema contains only the collections and entity types with access keys and identifiers, but no referential constraints. Additional implicit constraints can be discovered (schema refinement) through the procedural fragment analysis. The structure of `CUS-DESC` can be decomposed as `DESCRIPTION` because there is an assignment instruction (`move CUS-DESC to DESCRIPTION`) between `CUS-DESC` and `DESCRIPTION` and `DESCRIPTION` has a more precise decomposition (decomposed into two sub-variables) than `CUS-DESC` (only a long string of characters). A referential constraint can be discovered through the analysis of the second part of the code fragment: the user is asked for a `CUS-CODE`, if `CUS-CODE` exists in the `CUSTOMER` file (`read`) then it is copied into `ORD-CUS` and the `CUS` record is stored into the file.

Now we have the complete logical schema, the data structure conceptualization transforms the logical schema into a conceptual schema:

- *Preparation*

  Removing the indexes and collections. Before removing the collections, the entity types are renamed as the collections, because the collections have more meaningful names. A COBOL habit is to prefix fields by the name of the record to have unique names. So the common prefix of the attributes be removed.
- *Untranslation*

  Transforming the referential constraints into relationship type.
- *Conceptual normalization*

  Disaggregating the compound attribute (*CUS-DESC*) and renaming objects.

1. Orders that are not already paid.

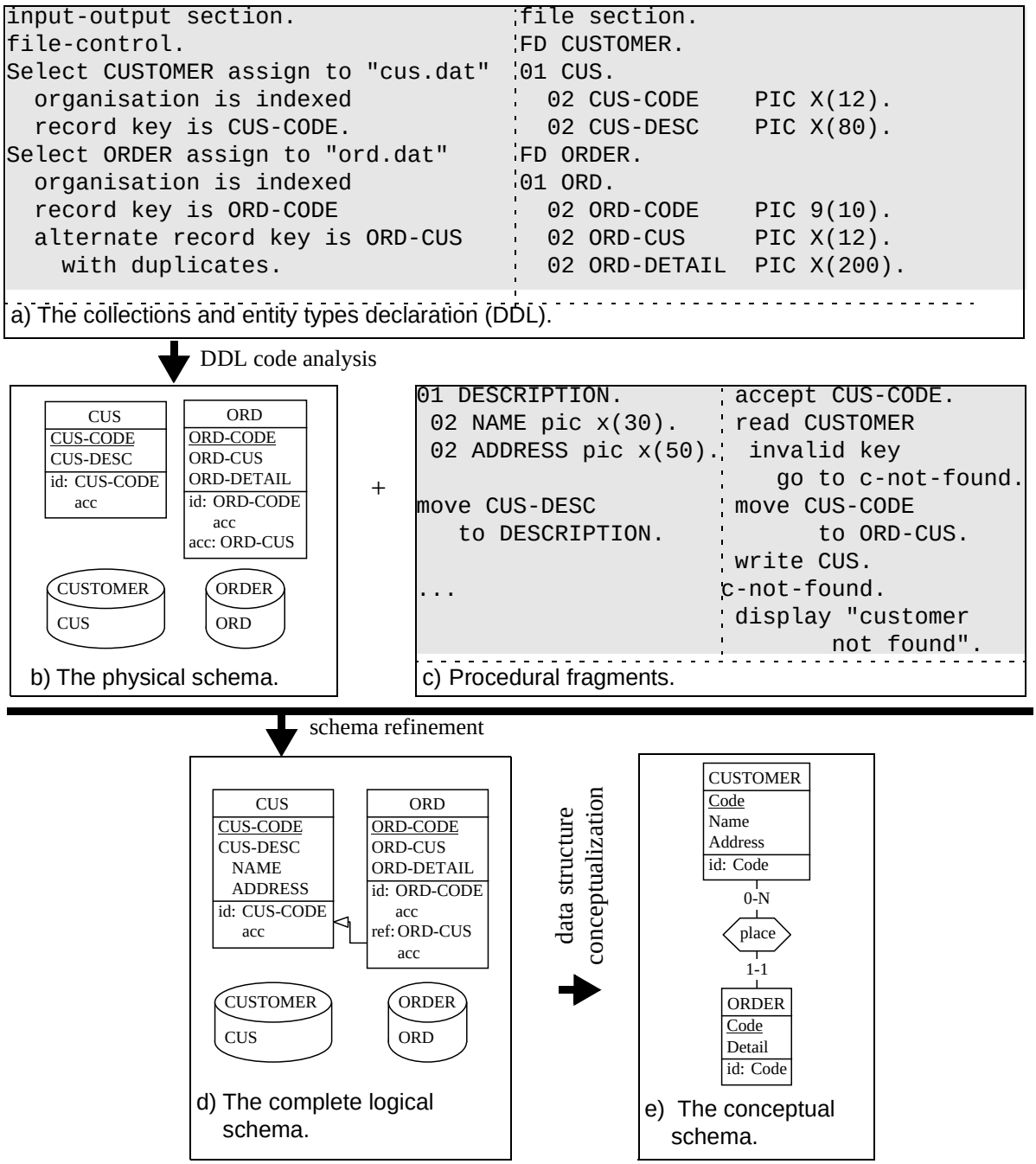


FIGURE 28. Database reverse engineering example.

CHAPTER 4 *Data structure extraction*

This chapter describes in detail the data structure extraction process. This process is divided in four steps. The first, the DDL code analysis, extracts from the data description language script the explicit structures and constrains in order to produce the raw physical schema. If more than one raw physical schemas exist, the physical integration step integrates them into a single schema. The schema refinement step enriches the integrated schema with explicit constraints revealed by the analysis of the source code, the data, etc. Finally, the schema cleaning step discards the physical constructs that are no longer needed. The main constraints that are searched for are described as well as the elicitation techniques that are used to recover the constraints during the schema refinement steps.

4.1. Introduction

The data structure extraction is the most crucial and difficult part of the DBRE. Data structure extraction analyzes the existing (legacy) system to recover the complete logical schema. This chapter details the data structure extraction processes and deeply analyzes the schema refinement process. The schema refinement recovers the implicit constructs, i.e. the constructs that are not explicitly declared in the DDL code. The implicit constructs that are looked for are enumerated and analyzed. The possible sources of information used to recover the data structure are presented. Techniques that can be used to recover implicit constructs through the analysis of the different source of information are presented. Those techniques are called elicitation techniques.

Finally, a schema refinement methodology is presented. This methodology is a repetitive process that searches for a possible missing constraint (called hypothesis) and tries to validate this hypothesis. This process is iterated until no new hypothesis is discovered. The conditions to decide when all the hypotheses have been discovered are discussed as well as how the elicitation techniques are used to discover and to validate the hypotheses.

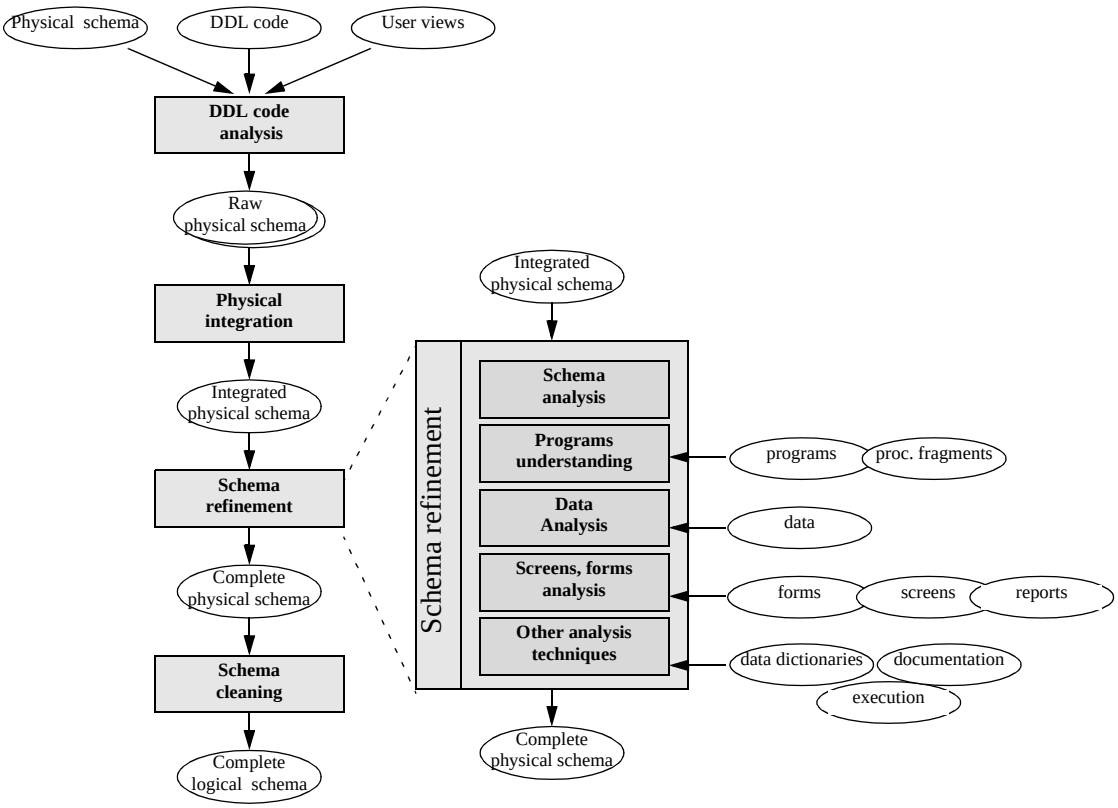


FIGURE 29. Data structure extraction.

4.2. The methodology

The main processes of the data structure extraction are the following (Figure 29).

- *DDL code analysis*

  It analyzes the physical schema, DDL code or user views, in order to extract the explicit constructs and constraints. It provides the raw physical schema. This schema contains all the data structures and constraints declared and only them.

- *Physical integration*

  When more than one source has been processed, several raw physical schemas can be available. All those schemas are integrated into a global one, giving the integrated physical schema.

- *Schema refinement*

  The integrated physical schema obtained so far is enriched with implicit constructs that are found through the analysis of the other sources of information as the procedural code, the screen layout, etc. The result of the schema refinement is called the complete physical schema. This schema is the memory of the whole data structure extraction phase; it contains all the data structures and constraints discovered. It contains the details about the physical implementation of the database (as indexes, page size, etc.) and the structure of the data.

COBOL statement	Physical abstraction
`select S assign to P`	collection S assigned to physical file P
`record key is F`	primary identifier with attribute F; and access key with field F.
`alternate record key is F`	attribute F is a secondary identifier and an access key.
`alternate record key is F with duplicates`	attribute F is an access key.
`fd S.` `01 R`	entity type R within storage S.
`05 F pic 9(n)`	numeric attribute F of size n, associated with its parent structure (entity type or compound attribute).
`05 F pic X(n)`	alphanumeric attribute F of size n, associated with its parent structure (entity type or compound attribute).
`05 F1.` `10 F2 ...`	compound attribute F1, with sub-attribute F2, etc.
`05 F ... occurs n times`	multivalued attribute F, with cardinality n-n.

FIGURE 30. Main abstraction rules for COBOL file structures.

SQL statement	Physical abstraction
`create dbspace S ...`	collection S.
`create table T (...) in S`	entity type T within collection S.
`F numeric(n)`	numeric attribute F with size n.
`F char(n)`	character attribute F with size n.
`... not null`	the current attribute as mandatory
`primary key (F)`	primary identifier with attribute(s) F
`... unique (F)`	secondary identifier with attribute(s) F
`foreign key (F) references T`	attribute(s) F a foreign key referencing entity type T.
`create index on T(F)`	access key with attribute(s) F.
`create unique index on T(F)`	secondary identifier with attribute(s) F; access key with attribute(s) F.

FIGURE 31. Main abstraction rules for relational structures.

- *Schema cleaning*

  Discards physical constructs that are no longer needed in order to get the complete logical schema (or simply the logical schema).

The complete logical schema includes all the data structures and constraints discovered during the data structure extraction. This schema may no longer be DMS compliant for at least two reasons. It is the result of the refinement process, which enhances the schema with recovered constraints, these

constraints are mainly constraints that cannot be expressed in the DMS. The complete logical schema describes the structure of the persistent data of the application and in some application more than one DMS is used. For example, some data are stored in COBOL files and others in a SQL database. If two DMS do not have the same data model, the resulting complete logical schema is compliant with none of the DMS.

On the other hand, this schema is still very close to the current implementation. It can be described as the programmer's view of the database with all the constraints and details needed to write or modify programs that access the database.

4.2.1. DDL code analysis

This process clearly is the simplest one in the data structure extraction. It consists in deriving physical abstractions from each DDL construct. The set of rules is easy to state in most DMS, provided the target abstract physical model includes a sufficiently rich set of features. It must be noticed that each DDL, even in the most modern DMS, includes clauses intended to declare physical concepts (e.g. indexes and clusters), logical concepts (entity types, attributes and referential constraints) as well as conceptual concepts (identifiers). Separating the DMS constructs in the standard abstraction levels is sometimes tricky (as in IMS for instance). Figure 30 and figure 31 show the main abstraction rules for converting COBOL and SQL-2 code into abstract physical structures. These conversion rules should be adapted to the specificity of each DMS. Similar rule sets can be defined for CODASYL, DL/1, TOTAL/IMAGE or OO data structures.

```
create table CUSTOMER(
 C_ID integer primary key,
 C_DATA char(80));

create table ORDERS(
 O_ID integer primary key,
 OWNER integer,
 foreign key(OWNER)
   references CUSTOMER);
```

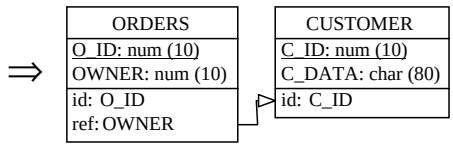


FIGURE 32. SQL-DDL and its physical abstraction.

Figure 32 shows a code fragment SQL-DDL and its corresponding physical abstraction.

Almost all CASE tools propose some kind of DDL code analysis for the most popular (generally relational) DMS. Some of them are able to extract relational specifications from the system data dictionary as well. Few can cope with non relational structures.

4.2.2. Physical schema integration

When more than one DDL source has been processed, the analyst is provided with several extracted schemas. Let us mention some common situations: DBD[1] and PSB[2] (IMS), schema and sub-schemas (CODADYL), file structures from each the application program (standard files), database

1. Database Description.
2. Program Specification Block.

schema, COBOL copy books (source code fragment that are included in the source program at compile time) etc. All these schemas have some common elements and each one may include specific elements not present in the other one. The final logical schema must include the specifications of all the partial views, through the *physical integration* process. This process differs from the approaches proposed in the literature on the integration of conceptual views ([Garcia et al.-1995]). In particular, we can identify three specific characteristics of physical schema integration.

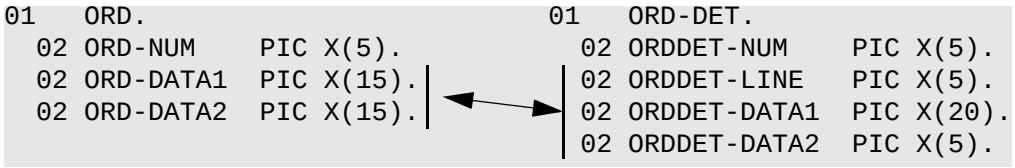


FIGURE 33. Example of incompatible record declarations.

1. Each physical schema is a view of a unique and fully identified physical object, namely the legacy database. Consequently, syntactic and semantic conflicts do not represent divergent user views but rather insufficient analysis. There is a syntactic conflict if two declarations of the same physical object are incompatible. For example figure 33 represents two COBOL declarations that are incompatible because the structure of one of them is not included in the other one. A semantics conflict occurs when the same physical object can represent different kind of information with a different semantics.

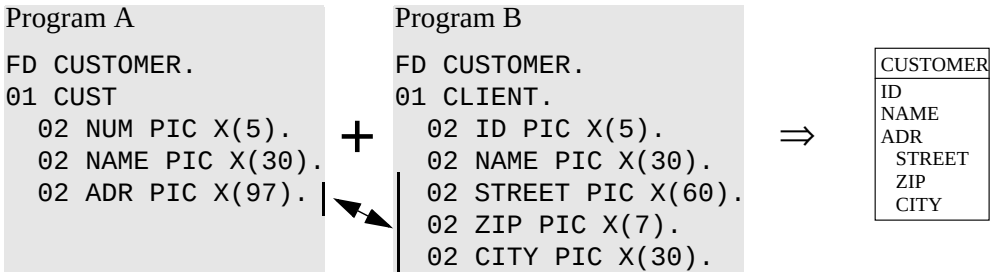


FIGURE 34. Example of integration based on data's offset and length.

2. Physical and technical aspects of the data can be used in correspondence heuristics, such as offset and length of data fields. For example, if two COBOL programs use the same physical file (same file on the disk) but the declarations of the corresponding record types are not exactly the same, the offset and length of the fields can be used to find a common representation as shown in figure 34.
3. There may be a large number of such views. For instance, a set of COBOL files serving a portfolio of 1000 programs units will be described by 1000 partial views. In addition, there is no global schema available for these files. The latter will be recovered by integrating these views.

This process integrates only the elements of the different raw physical schema that are different views of the same physical object, e.g. the same file on the disk. This process does not integrate different physical objects that have the same semantic. For example, in an hospital, the health care system records administrative information about the people in the *patient* entity type and the invoice system records these information in the *customer* entity type. These two entity types are not integrated during this process, but will be integrated during the conceptualization process.

4.2.3. Schema refinement

The main problem of the data structure extraction phase is to discover and to make explicit, through the refinement process, the structures and constraints that were either implicitly implemented or merely discarded during the development process. The variety of implicit constructs can be very large; the main implicit structures and constraints we are looking for are the following: entity type and attribute disaggregation, identifier, referential constraints, functional dependency, meaningful names, etc.

This process will be developed in section 4.8.

4.2.4. Schema cleaning

The complete logical schema includes all the data structures and constraints discovered during the data structure extraction. It is still very close to the current implementation to describe the programmer view of the database. To fulfil those criteria, the complete physical schema cannot be deeply reorganized to obtain the complete logical schema. For example, an object cannot be renamed, otherwise the programmer could not write applications that access the database. Redundancies are noted but cannot be suppressed, otherwise when the programmer modifies the program he will forget to maintain the redundancy.

The constructs that can be modified or suppressed during the schema cleaning depend on the DMS used. For example, for COBOL files, the indexes are needed, because to access the data, the programmer has to specify which index to use. On the other hand, for a relational DMS, indexes can be discarded because the programmer does not need to know the indexes to write programs that access the data (queries).

```
create table CUSTOMER(
  C-ID integer primary key,
  C-DATA char(80))
create table ORDER(
  O-ID integer primary key,
  OWNER integer
  foreign key(OWNER)
references CUSTOMER)
```

FIGURE 35. Tables declaration with explicit foreign key.

```
create table CUSTOMER(
  C-ID integer primary key,
  C-DATA char(80))
create table ORDER(
  O-ID integer primary key,
  OWNER integer)
```

a) Table declaration without foreign key.

```
exec SQL
  select count(*) in :ERR-NBR from ORDER
    where OWNER not in
      (select C-ID from CUSTOMER)
end SQL
...
if ERR-NBR > 0 then
  display ERR-NBR,
     'referential constraint violation';
```

b) Procedural code fragment that verifies the validity of the foreign key.

FIGURE 36. Tables creation and implicit foreign key implementation.

4.3. Explicit/implicit constructs

An *explicit construct* is a component or a property of a data structure that is declared through a specific DDL statement. An *implicit construct* is a component or a property that holds in the data structure, but that has not been declared explicitly. In general, the DMS is not aware of implicit constructs, though it can contribute to its management (through triggers for instance). The analysis of the DDL statements alone leaves the implicit constructs undetected. The most popular example certainly is that of referential constraint, which we will use to explain this point. Let us consider the code of figure 35 in which two tables, linked by a foreign key, are declared. This foreign key is an explicit construct because a specific DDL statement has been used to declare it. On the other hand, the code of figure 36.a is the declaration of two tables in which no referential constraint has been declared and the code of figure 36.b represents a fragment of the application that strongly suggests that column OWNER is expected to behave as a referential attribute. If the analyst is convinced that this behavior must be taken for an absolute rule, then OWNER is an implicit referential attribute.

The problem is much more complex for standard files, for which no computerized description of their structures and constraints exists in most cases. The analysis of each source program only provides a partial view of the collection and entity type structures. All the other constraints need to be represented and checked by other means such as procedural sections. Unfortunately, these practices are also common in (true) databases, where all the expressiveness of the DMS is not necessarily used. For example, in the above example (figure 36), the referential constraint was not declared in the DDL although it would have been possible.

By examining the expressive power of DMS, compared with that of semantics representation formalism, and by analyzing how programmers work, we can identify five major sources of implicit constructs.

- *Structure hiding*

  Structure hiding concerns a source data structure or constraint S1, which could be implemented in the DMS. It consists in declaring it as another data structure S2 that is more general and less expressive than S1. In COBOL applications for example, a compound/multivalued attribute, or a sequence of contiguous fields can be represented as a single-valued atomic attribute (e.g., a filler). In a CODASYL or IMS database, a one-to-many relationship type can be implemented as a many-to-many link, through a record/segment type, or can be implemented by an implicit reference attribute. In an SQL-2 database, some referential constraints can be left undeclared by compatibility with older DMS (see the previous example). The origin of structure hiding is always a decision of the programmer, who tries to meet requirements such as attribute reusability, genericity, program conciseness, simplicity, efficiency. Structure hiding can also be the result of bad practices such as poor programming practice, straightforward transformation of a legacy system, disorganization that results from prolonged maintenance as well as consistency with legacy components of the application.
- *Generic expression*

  Some DMS offer general purpose functionalities to enforce a large variety of constraints on the data. For instance, current relational DMS propose column and table check predicates, views with check option, triggers mechanisms and stored procedures. These powerful techniques can be used to program the validation and the management of complex constraints in a centralized way. The problem is that there is no standard way to cope with these constraints. For instance, constraints such as referential integrity can be encoded in many forms, and their elicitation can prove much more complex than for declared foreign keys.

- *Non declarative structures*

  Non declarative structures have a different origin. They are structures or constraints that cannot be declared in the target DMS, and therefore are represented and checked by other means, external to the DMS, such as procedural sections in the application programs or in the user interface. Most often, the checking sections are not centralized, but are distributed and duplicated (frequently in different versions), throughout the application programs. For example, standard files commonly include referential constraints, though current DMS ignore this construct. In the same way, CODASYL DMS do not provide explicit declaration of one-to-one relationship types, which often are implemented as (one-to-many) set types and integrity validation procedures.

- *Environmental properties*

  In some situations, the environment of the system guarantees that the external data to be stored in the database satisfy a given property. Therefore, the developer has found useless to translate this property in the data structure, or to enforce it through DMS or programming techniques. Of course, the elicitation of such constraints cannot be based on data structure and program analysis. For example, if the content of a sequential file comes from an external source in which uniqueness is guaranteed for one of its attribute, then the database file inherits this property, and an identifier can be asserted accordingly.

- *Lost specifications*

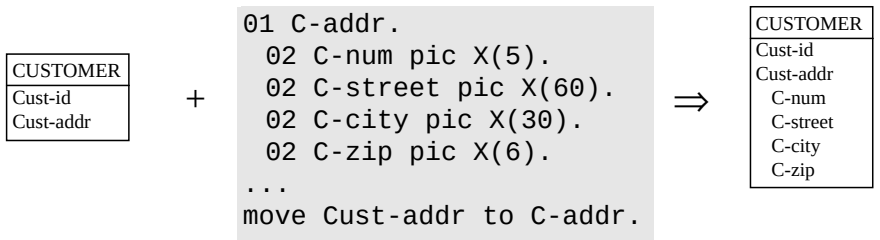
  Lost specifications correspond to facts that have been ignored or discarded, intentionally or not, during the development of the system. This phenomenon corresponds to flaws in the system that can translate into corrupted data. However, lost specifications can be undetected environmental properties, in which case the data generally are valid.

4.4. Implicit structures and constraints

This section describes some of the main implicit structures and constraints that can be found in actual reverse engineering projects of various size and nature. It is important to keep in mind that this analysis is DMS-independent. Indeed, almost all the patterns that will be discussed have been found in practically all the types of database.

- *Finding the fine-grained structure of entity types and attributes.*

  An attribute, or an entity type, declared as atomic, has an implicit decomposition, or is the concatenation of contiguous independent attributes. The problem is to recover the exact structure of this attribute or of this entity type. This pattern is very common in standard files and IMS databases, but it has been found in modern databases as well, for instance in relational tables.



a) The raw entity type. b) Some code fragment. c) The refined entity type.

FIGURE 37. Example of an attribute refinement.

For example, the extraction of the DDL gives the raw entity type of figure 37.a. There is some procedural code that assigns an attribute to a variable with a finer decomposition (figure 37.b). So that the schema can be modified to obtain the refined entity type (figure 37.c).

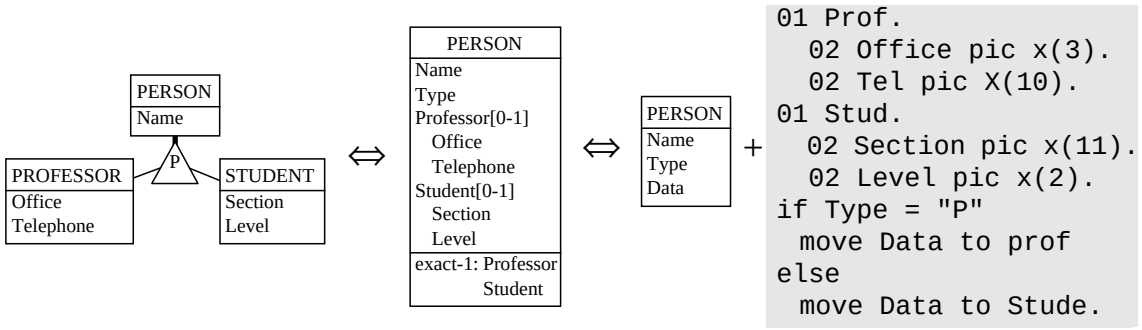


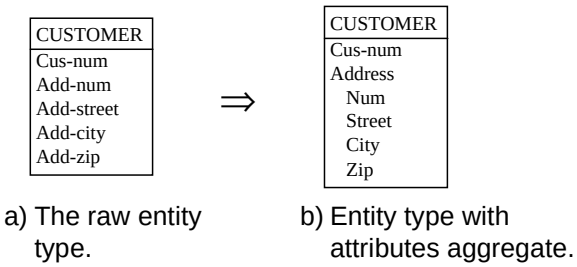
FIGURE 38. Example of optimization that merge two attributes into one.

The attribute and the target variable may have incompatible structures, such that one of the structures is not included in the other one. There are three reasons for this incompatibility. The first one is that there is an error in the program. The second one is an misunderstanding (error) in the analysis of the information sources. The third one is that the attributes (or entity type) have more than one usage, which is a common optimization technique. In legacy systems, the number of files that can be opened simultaneously was limited, so to reduce the number of files used by an application, different entity types were stored in the same file. For example, order headers and order details were stored in the same file with the order details following their order header. Another advantage was that it was easy and fast to access the details when the header was found. Another optimization to save disk space used by an entity type is to use the same physical location to store two attributes. This is possible if the two attributes are never present together (exactly-one constraint). An attribute is added (`Type` in figure 38) to the entity type to know which value is stored in the common attribute (`Data`). `Type` is tested (in the procedural code) to know how to interpret the value contained into the `Data` attribute.

- *Finding optional (nullable) attributes*

  Most DMS postulate that each attribute of each entity type has a value. In general, giving an attribute no value consists in giving it a special value, to be interpreted as missing or unknown value. Since there is no standard way to implement this constraint, it must be discovered through, among others, program and data analysis.

  Usually the programmer uses the high-value or low-value to represent the Null in a numeric attribute and he fills alphanumeric attributes with space or some special character to represent an alphanumeric Null value.



a) The raw entity type.

b) Entity type with attributes aggregate.

FIGURE 39. Example of an attribute aggregate.

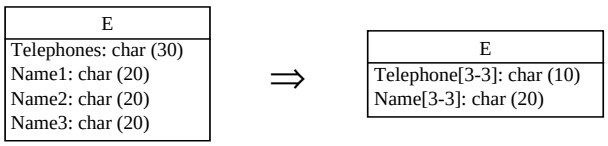
- *Finding attribute aggregates*

  A sequence of seemingly independent attributes are originated from a source compound attribute which was decomposed. The problem is to rebuild this source compound attribute. This is a typical situation in relational, RPG, IMS and TOTAL/IMAGE databases that impose flat structures.

  For example, figure 39.a contains attributes *Add-num*, *Add-street*, *Add-city* and *Add-zip* that are obviously the different parts of an address. They have the same prefix (*Add-*) and domain knowledge tells us that an address is composed of a number, a street, a city and a zip code. They can be aggregated as the *Address* compound attribute (figure 39.b).

- *Finding multivalued attributes*

  An attribute, declared single-valued, appears to be the concatenation of the values of a multivalued attribute. Or a list of single valued attributes of the same type and same length appear to be the materialization of a multivalued attribute. The problem is to detect the repeating structure, and to make the multivalued attribute explicit. Relational, RPG, IMS and TOTAL/IMAGE databases, that cannot represent multivalued attribute, commonly include such constructs.



a) The raw entity type.

b) The entity type with multivalued attributes.

FIGURE 40. Example of single-valued and list of single valued attributes transformed into multivalued attributes.

  Telephones (figure 40.a) is 30 characters long. The associated code shows that it is split into an array. So *Telephones* can be transformed into *Telephone[3-3]* of 10 characters (figure 40.b). In figure 40.a, the list of attributes *Name1*, *Name2* and *Name3* can be transformed into a multivalued attribute (*Name* in figure 40.b).

- *Finding multiple attribute and entity type structures*

  The same attribute, or entity type structure, can be used as a mere container for various kinds of value.

  For instance, a *CONTACT* entity type appears to contain entity types of two different types, namely *CUSTOMER* and *SUPPLIER*.

- *Finding entity type identifiers*

  The identifier (primary or secondary) of an entity type is not always declared. Such is the case for sequential files or CODASYL set types.

  An example of a pattern used to detect that there is an undeclared identifier in a sequential field, is that the user is asked for the name of the customer. Then there is a loop that goes through the file and the loop is exited when the first customer with the given name is found and no other customer with that name is looked for.

- *Finding identifiers of multivalued attributes*

  Structured entity types often include complex multivalued compound attributes. Quite often too, these values have an implicit identifier.

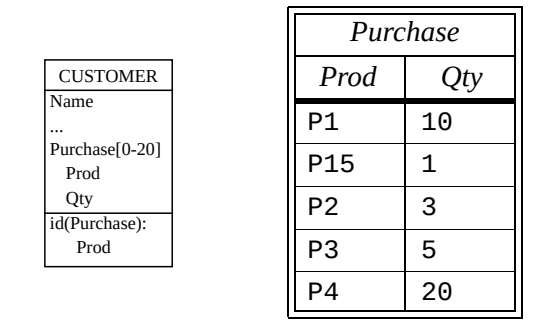


FIGURE 41. Example of identifier of a multivalued attribute.

For instance (figure 41), in each *CUSTOMER* entity type, there are no two *PURCHASE* compound values with the same *PRODUCT* value.

- *Finding referential constraints*

  In multi-file applications, there exist inter-file links, represented by referential constraints, i.e., by attributes whose values reference an entity type in another collection. The most common form of referential constraint (called standard foreign key) is made of one or several mandatory attributes. It targets the identifier of an entity type and both ends of the referential constraint are defined on the same domain. In legacy systems there are not only standard foreign keys, but a lot of tricky patterns such as optional, recursive or computed foreign key. For a complete discussion about non standard foreign keys see [Hainaut-1997a].

  For example, figure 32 shows an implicit referential constraint that is validated by some procedural code.

```
read CUS
 invalid key go to Err.
move Cus-id to Ord-cus.
move Cus-name to Ord-cus-name.
...
write ORDER.
```

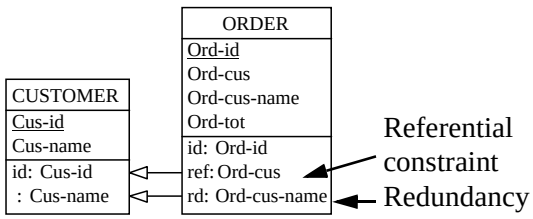


FIGURE 42. Example of redundancy parallel to a referential constraint.

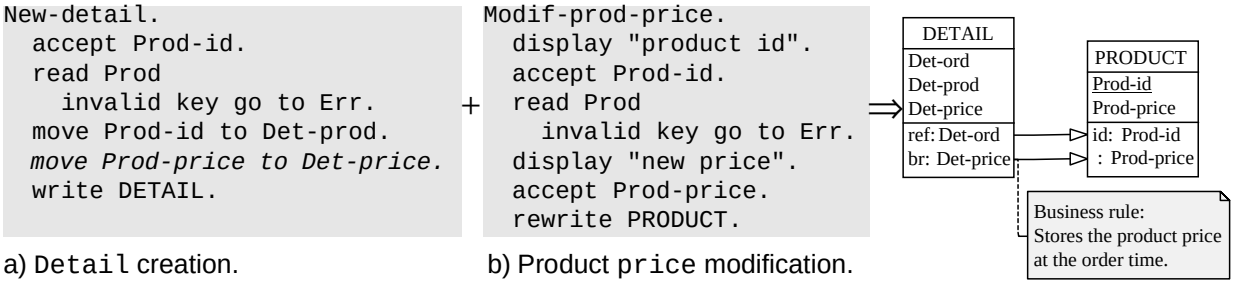
- *Finding functional dependencies*

  As commonly recognized in the relational database domain, normalization is a recommended property. Thus functional dependencies should not be found in ideal applications. However, many actual databases include unnormalized structures, generally to get better performance. Functional dependencies save disk access and computation but increase disk space needed and complexity of the management rules that maintain the coherence of the data. For example (figure 42), storing the name of the customer (*Ord-cus-name*) and the total amount of the order (*Ord-tot*) into *ORDER* entity type saves disk access to get the name of the customer and cpu time to compute the total amount of the order when the invoice is printed. The drawbacks of this optimization is when the list of the ordered products is updated, *Ord-tot* must be computed. When the name of the customer changes (spelling error), all his orders need to be updated.

  Redundancies are a special kind of functional dependencies where the function is the identity. The value of the origin field is copied into the target one.

Functional dependencies are usually parallel to a referential constraints. In order to know which entity type is the origin of the dependency, the program follows a referential constraint. For example, in figure 42, there is a referential constraint from `ORDER` to `CUSTOMER` and `Cus-name` is copied into `Ord-cus-name` (redundancy).

Be aware of the conclusion. If during the analysis of a program fragment a dependency is detected between two attributes, it does not automatically mean that this data dependency is always verified. This data dependency can be true at some moment but not all the time. This can be seen as a business rule and must be documented.



a) `Detail` creation. b) Product `price` modification.

FIGURE 43. An example of business rule.

Figure 43 gives an example of a very simple business rule. Each time a product is ordered, its price (`PRODUCT.Prod-price`) is copied in the `DETAIL.Det-price` record (figure 43.a), so it can be concluded that there is some data dependency. The analysis of product price (`Prod-price`) modification (figure 43.b) shows that when the price of the product is changed for a `PRODUCT` this change is not propagated to `DETAIL.Det-price`. It can be concluded that there is no data dependency between `Det-price` and `Prod-price` as firstly assumed. The price stored in `Det-price` is the price of the product at the order time and not the current price of the product. This price is kept to compute the amount of the order at the price of the order time when the invoice will be printed (perhaps several weeks later).

It is important to make the difference between data dependencies and business rules, because attributes that are data dependent are removed, during the conceptualization process, whereas the attributes that take part in a business rule must be kept.

- *Finding sets behind arrays*

  Multivalued attributes are generally declared as arrays, because the latter is the most obvious, if not the only construct available in host languages and DMS to store repeating values. Unfortunately, an array is a much more complex construct than a set. Indeed, while a set is made up of an unordered collection of distinct values, an array is a storage arrangement of partially filled, indexed cells that can accommodate non distinct values. In short, an array basically represents ordered collections of non distinct values with possible holes (empty cells). For each array, one must answer three questions: are the values distinct? Is the order significant? What do holes mean? Clearly, usage pattern and data analysis are the key techniques to get the answers.

- *Finding exact minimum cardinality of attributes and relationship types*

  Multivalued attributes declared as arrays, have a maximum size specified by an integer, while the minimum size is not mentioned, and is under the responsibility of the programmer. For instance, attribute *DETAIL* has been declared as "`02 DETAIL OCCURS 20`", and its cardinality has been interpreted as [20-20]. Further analysis has shown that this cardinality actually is [1-20].

- *Finding exact maximum cardinality of attributes and relationship types*

  The maximum cardinality can be limited to a specific constant due to implementation constraints. Further analysis can show that this limit is artificial, and represents no intrinsic property of the problem. For instance, an attribute cardinality of [0-100] has been proved to be implementation-dependent, and therefore relaxed to [0-N], where N means unlimited.

- *Finding existence constraints*

  Sets of attributes and/or roles can be found to be coexistent, that is, for each entity type, they all have a value or all are null. There are other similar constraints, such as exclusive (at most one attribute is not null) and at least one (at least one attribute is not null). These constraints can be the only trace of embedded attributes aggregates or of sub-type implementation.

  For example in the *CUSTOMER* entity type there are *Wedding-date* and *Spouse-name* attributes. It can be discovered that there is a coexistence constraint between these two attributes.

- *Finding enumerated value domains*

  Many attributes must draw their values from a limited set of predefined values. It is essential to discover this set.

  A typical enumerated attribute is the *Sex* attribute that has `{M, F}` as domain value.

- *Finding constraints on value domains*

  In most DMS, declared data structures are very poor as far as their value domain is concerned. Quite often, though, strong restriction is enforced on the allowed values.

  For example the ordered quantity of a product is a strict positive (>0) number but it is store into a numeric attribute that allows negative value to be stored.

- *Finding meaningful names*

  Some programming disciplines, or technical constraints, impose the usage of meaningless names, or of very condensed names whose meaning is unclear. On the contrary, some applications have been developed with no discipline at all, leading to poor and contradictory naming conventions. During data structure extraction, objects cannot be renamed, otherwise the complete logical schema will not represent the current implementation. If more meaningful names are found, they are noted in order to be able to rename the object during the data structure conceptualization process.

  For example, an attribute named *Cus-F2* that is used to fill a screen field that is labeled "customer name". It is noted that the attribute *Cus-F2* represents the name of the customer and during the data structure conceptualization it will be renamed *Name* or *Cus-Name*.

4.5. The information sources

To discover an implicit construct, the analyst generally cannot limit his analysis to one information source. On the contrary he has to rely on all the possible information sources, such as: application programs, data, HIM procedural fragments, screen and report layout, generic DMS code fragments[1], existing documentation, interviews, domain knowledge, operation environment knowledge, etc.

1. Some DMS offer general functionality to enforce a large variety of constraints on the data.

The analyst needs to analyze several of those sources because none of them contains all the hints for all the constraints. For example, some constraints are not implemented in the application program because they are verified by some environmental properties (the input data are always correct, they come from another fully reliable application). Constraints are not discovered by the data analysis because there is some erroneous data. On the other hand, spurious constraints can be discovered in the data because the set of data is too small, for example, an attribute is an identifier.

The most common sources are:

- *DMS-DDL (schemas and views)*

  This is the database declaration statements, which specify the explicit structures and constraints. The database can also contain some procedural code fragments, trigger, check or stored procedure, that need to be analyzed to discover some implicit constraints (see generic DMS code fragments below).

- *Data dictionary/physical schema*

  The data dictionary contains the description of the actual state of the database structures and constraints. Usually it is updated by the DMS itself and is the most up-to-date information source. As the DMS-DDL statements, it contains the explicit data structures but also some procedural code.

- *Generic DMS code fragments*

  Modern databases may include code sections that monitor the behavior of the database. Check/assertion predicates, triggers and stored procedures are the most frequent. They generally express in a concise way the validation of data structures and integrity constraints. As any code, they are less easy to analyze since there is no standard way to code a specific integrity constraint. They implement implicit constraints.

- *Application source code*

  Many data structures and constraints that are not explicitly declared are coded, among others, as procedural sections of the programs. For this reason, one of the most important information sources is the application program.

- *Screen and report layout*

  A screen form or a structured report can be considered to be derived views of the data. The layout of the output data as well as the labels and comments can bring essential information on the data.

- *Current documentation*

  In some reverse engineering projects, there is some kind of documentation available. Though these documents are often partial, obsolete and even incorrect, they can bring useful information. Of course, the comments that programmers include in the programs can also be a rich source of information. Most DMS allow administrators to add short comment to each schema object.

- *External data dictionaries and CASE repositories*

  Third-party or in-house data dictionary systems allow data administrators to record and maintain essential descriptions of the information resources of an organization, including the file and database structures. They can provide informal but very useful description of the data with which one can better grasp their semantics. The main problem with these sources is that they generally have no automatic 2-way link with the databases, and therefore may include incomplete, obsolete or erroneous information. The same can be said of CASE tools, which can record the description of database structures at different abstraction levels. While such tools can gener-

ate the database definition code, they generally offer no easy way to propagate direct database structure modifications into these schemas.

- *Domain knowledge*

  It is inconceivable to start a reverse engineering project without any knowledge on the application domain. Indeed, being provided with an initial mental model of the objectives and of the main concepts of the application, the analyst can consider the existing system as an implementation of this model. The objective is then to refine and to validate this first-cut model. This is why the analyst must have some deep domain knowledge or he must be in tight contact with a client analyst that is assigned to the reverse engineering project. In this context, interviewing regular or past users, developers or domain knowledge experts can be a fruitful source of information, either to build a first domain model, or to validate the model elaborated so far.

- *Data*

  The data themselves can exhibit regular patterns, or uniqueness or inclusion properties that provide hints that can be used to confirm or disprove structural hypotheses. The analyst can find hints that suggest the presence of identifiers, referential constraints, attribute decomposition, optional attributes, functional dependencies, existence constraints, or that restrict the value domain of an attribute for instance.

- *Non-database sources*

  Small volumes of data can be implemented with general purpose software such as spreadsheet and word processors. In addition, semi-structured documents are increasingly considered as a source of complex data that also need to be reverse engineered. Indeed, large text databases can be implemented according to representation standards such as SGML, XML or HTML that can be considered as special purpose DDL.

- *Program execution*

  The dynamic behavior of a program working on the data gives information on the requirements the data have to meet to be recorded in the files, and on links between stored data. In particular, combined with data analysis, filled-in forms and reports provide a powerful examination means to detect structures and properties of the data.

- *DMS logs*

  Some DMS store in a log all the data access or queries performed with some statistics. The analysis of such log can be interesting to know which queries are performed, specially for DMS with powerful query language such as SQL databases.

- *Environment properties*

  The environment properties, as the DMS, the development tools, development language, programming principle, the hardware used can imply some non functional requirements. Those requirements can influence the way some constraints are verified or simply discarded.

- *Application history*

  Who are the analysts who write and maintain the application, what are the different DMS and programming languages used? This information can explain the techniques used to code some constrains and data structures. For example, if the application was migrated from flat files to a relational DMS, typical file structures are found and not relational one.

- *Corporate practice*

  Some corporate have an in-house methodology and habit. Their knowledge can ease the data structure extraction. For example, all the names of the identifier start by the keyword ‘id’, or at the beginning of each procedure there is a comment describing its goal.

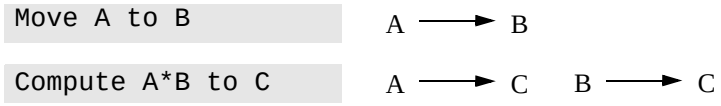


FIGURE 44. Some common dataflow.

4.6. Elicitation techniques

Though there exists a fairly large set of potentially implicit constructs and information sources, there is a limited set of common analysis techniques. We describe some of them.

- *Dataflow analysis*

  Examining in which variable data values flow in the program can put into light structural or intentional similarities between these variables. For instance, if variable *B*, with structure *Sb* receives its values from variable *A*, with structure *Sa*, and if *Sb* is more precise than *Sa* (*Sb* has a finer decomposition), then *A* can be given structure *Sb*. The term flow must be taken in a broad sense: if two variables belong to the same path in the dataflow, at some time, and in some determined circumstances, their values can be the same, or one of them can be a direct function of the other. Figure 44 presents some common dataflow applied to the COBOL language.

  More sophisticated, or less strict relations can be used as `if A=B` and `Compute A*B to C`. Such patterns do not define equality of values between `A` and `B`, but rather a certain kind of similarity. This dependency could imply that `A` and `B` have compatible value domains. We are not only interested in direct relations between variables but also in transitive relations. For example, `Move A to B` and `Move B to C`, imply the relation between `A` and `B` and between `B` and `C`, but also the transitive relation between `A` and `C`.

- *Programming clichés*

  Disciplined programmers carefully use similar standard patterns to solve similar problems. Therefore, once the pattern for a definite problem has been identified (called programming cliché), searching the application programs or other kind of procedural fragments for instances of this pattern allows us to locate where problems of this kind are solved [Henrard et al.-1998a], [Petit et al.-1994], [Quilici et al.-1997] and [Signore et al.-1994].

```
Accept @org_id .
...
Read @orgin Key Is @org_id
  Invalid Key Go To @error_label
...
Move @org_id to @targ_ref.
...
Write @target.
```

FIGURE 45. A cliché to detect a referential constraint.

For example, we can write a cliché (see figure 45) that matches one of the many algorithms to validate the presence of a referential constraint in a COBOL program. Finding all the instances of this cliché in a huge COBOL program (one million LOC, split in 100 text sources) can be quickly done. Each time an instance of the cliché is found there is a strong evidence that there is a referential constraint. Actual data names have been replaced with cliché variable names (pre-

fixed with @), the dots (. . .) represent any instructions and the italic words are the reserved words of the language.

```
  FD CUSTOMER.
  01 CUS.
    02 CUS-NUM PIC 9(3).
    02 CUS-NAME PIC X(10).
    02 CUS-ORD PIC 9(2) OCCURS 10.
       ...
  01 ORDER PIC 9(3).
       ...
1 ACCEPT CUS-NUM.
2 READ CUS KEY IS CUS-NUM.
3 MOVE 1 TO IND.
4 MOVE 0 TO ORDER.
5 PERFORM UNTIL IND=10
6   ADD CUS-ORD(IND) TO ORDER
7   ADD 1 TO IND.
8 DISPLAY CUS-NAME.
9 DISPLAY ORDER.
```

a) COBOL program P.

```
  FD CUSTOMER.
  01 CUS.
    02 CUS-NUM PIC 9(3).
    02 CUS-NAME PIC X(10).
1 ACCEPT CUS-NUM.
2 READ CUS KEY IS CUS-NUM.
8 DISPLAY CUS-NAME.
```

b) Slice of P with respect to `CUS-NAME` and line 8.

FIGURE 46. Example of program slice.

- *Program slicing*

  This very powerful technique provides extracts (slices) from a large program according to defined criteria [Weiser-1984]. Considering program *P*, a point *p* in *P* (e.g. an instruction) and an object *V* (a variable or a record), the backward program slice of *P* with respect to the *slicing criterion <p, V>* is the set of all statements of *P* that can contribute to the state of *V* at point *p*. In other words, executing *P* and executing the slice give *V* the same value whatever the external conditions of the execution (section 6.4). Figure 46.b is the slice of the program of figure 46.a with respect to variable `CUS-NAME` and line 8.

  This technique allows the analyst to reduce the search space when he looks for definite information in large programs.

- *Names analysis*

  Experimented programmers carefully chose the names they give to the entity types, attributes and variables to ease program development and maintenance. They give meaningful names that suggest the semantics and the function of the objects. The analysis of the name of the objects can give very useful hints about the semantics and the structure of the data. In addition, this analysis can detect synonyms (several names for the same object) and homonyms (same name for different objects). Attributes called *Total_Amount*, *Rebate*, *Shipment_cost* or *Average_Salary* could be derived attributes since they suggest values that usually are computed or extracted from reference attributes. Names can also include important meta-data, such as structural properties (attribute names *Add-City-Name*, *Add-City-Zipcode* suggest a 3-level hierarchy), data type (*Integer-Level*), unit (*Volume-Tons*), language (*Title-engl*, *Title-germ*).

  For example, a common usage is to prefix or suffix the identifier by one of the following keywords: *id*, *code*, *num*, etc. Quite often the name of a reference attribute suggest the name of the target attribute or entity type.

- *Physical structure*

  Some physical structures (address alignment, entity type offset, abnormally long attributes, access keys, clusters, multi-record-types fields, etc.) may give hints on possible logical structures and constraints.

  For example, identifiers are usually supported by a (unique) access key because objects are accessed through their identifier. To optimize this access, the programmer usually declares an access key. Another example is that abnormally long attributes are candidates to refinement into compound or multivalued attributes.

- *Screen and reports analysis*

  Screen and reports are used to present data and/or let users modify them. They are views of the data, therefore, their structure generally gives important hints on the structure and semantic of the data they transmit. Frequently, one screen panel includes data from several entity types. Three kinds of information can be derived from the examination of screens and reports:

  - *Spatial relationships between data fields.*

    The way the fields are located on the screen may suggest implicit relationships.

  - *Labels and comments included in the panel*

    They bring information on the meaning, the role, meaningful name and the constraints of each screen field.

  - *Discarded attributes.*

    An attribute that does not appear on the screen can lead to several conclusions:

    - The attribute is an obsolete attribute that is not used anymore.
    - The attribute can also be optional and has no value in this context.
    - This attribute is redundant with another one that is already displayed in the form.
    - This may mean that this attribute designates an information that is given by the context, for instance about the customer of the order.

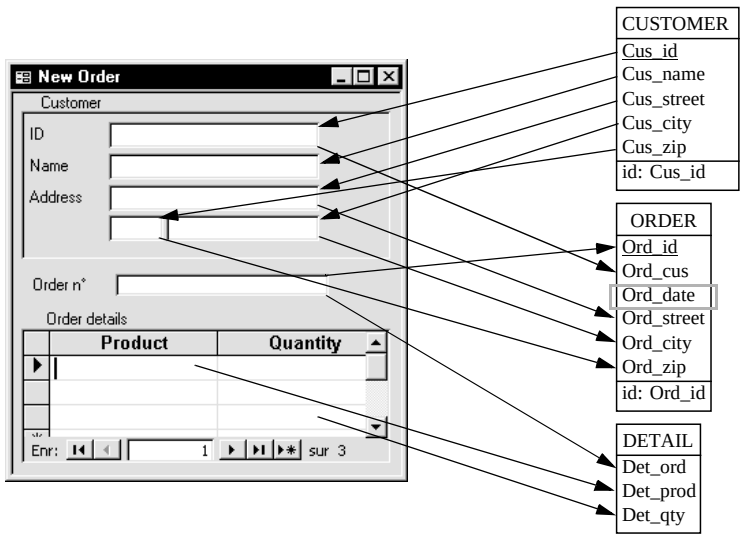


FIGURE 47. Dataflow between a dialog box and the entity types.

A screen layout can be examined as a stand-alone component, as suggested above. It can also be analyzed as source/target data structures of the programs that use it to communicate with their environment. Figure 47 shows a dialog box, used to enter a new order, with its associated entity

types. The dataflow between the dialog box and the entity type is represented by arrows. There are two sub panels that contain respectively a customer and the list of the order details. Each of them is associated with an entity type. They can be analyzed as follow:

•*Cus_id* → ID → *Ord_id*

Cus_id is the identifier of *CUSTOMER* and it is copied into *Ord_cus*. So there may be a referential attribute from *Ord_cus* to *Cus_id*.

•(*Cus_street, Cus_city, Cus_zip*) → Address → (*Ord_street, Ord_city, Ord_zip*)

the three attributes are grouped under the same label (*Address*), so they can be aggregated. *Ord_street, Ord_city, Ord_zip* are redundant attributes, they are copies of the corresponding *CUSTOMER* attributes that are "parallel" to the referential constraint.

•Order n° → *Ord_id* and *Det_ord*

suggests a referential constraint between *Det_ord* and *Ord_id*.

•Product → *Det_prod* and Quantity → *Det_qty*

more meaningful names can be noted for these two attributes.

•*Ord_date*

It does not receive its value from the dialog box. The source code must be analyzed to notice that it receives its value from the system (the date of the day).

The refined schema is shown in figure 48.

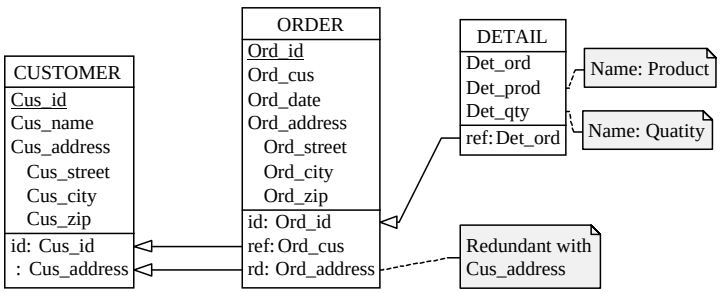


FIGURE 48. The refined schema.

Data reports can be considered both as data structures and as populated views of the persistent data. The first aspect is quite similar to that of screen layout: a report is a hierarchical data structure that makes relationships between data explicit. The second one relates to the data analysis heuristics.

- *Data analysis*

  Through the analysis of the contents of files and databases, some properties may be discovered or some hypotheses can be supported, proved or disproved.

  For example, if *Order.O-cust* is a referential attribute to *Customer*, then the referential integrity should be satisfied and each of its values must identify a *Customer* record. The following SQL query will check this condition by computing the number of violations:

```
select count(*)
from Order
where O-cust not in (select Cid
                     from customer)
```

However, the result, *n*, returned by this query must be interpreted with caution, because several conclusions can be drawn from it, depending, among others, on the size of the data sample which was analyzed:

Outcome	Interpretation
$n = 0$	**1.** *O-cust* is a referential attribute. **2.** Statistical accident, tomorrow, the result may be different. *O-cust* is not a referential attribute.
$0 < n < \varepsilon$	**1.** *O-cust* is not a referential attribute. **2.** *O-cust* is a referential attribute, but the query detected data errors. **3.** *O-cust* is a conditional referential attribute[a].
$0 \ll n$	**1.** *O-cust* is not a referential attribute. **2.** *O-cust* is a conditional referential attribute.

a. A conditional referential constraint is a referential constraint that is not always verified. It is only verified when some condition is satisfied: the reference attribute is null or another attribute has a given value.

- *Program execution analysis*

  The principle is to analyze the reactions of the program to selected stimuli, for instance, in terms of acceptance and rejection of input data and update queries. A running program also populates the screen panels and printed reports. So it is strongly linked with screen and report layout analysis.

- *Documentation analysis*

  When it still exists, and when it can be relied on, the documentation is the first information source to use. Normally, collection structures, attribute description and particularly their roles (such as referencing) should be documented. Before being used, the quality of the documentation should be assessed. Is it up-to-date? Does it describe the current version of the system or a previous one? Which formalism does it use?

- *Schema analysis*

  The analysis of working schema, i.e., the schema that is currently refined, can give hints about which constraints still need to be found. For example, if an entity type is not connected to the rest of the schema, with referential constraints or relationship types, it can suggest that there must be some missing referential constraint that has this entity type as target or origin. The schema analysis is quite easy to do because the schema is an abstraction of the database structure and it is stored in a CASE tool that offers analysis facilities.

4.7. The conflicts

During the data structure extraction some conflicts or inconsistencies can arise. These conflicts can have three different origins:

1. The same physical structure in the database can be used to materialize different logical concepts. For example, the same character string can contain the address of the customer and at another moment the label of the product, because the string is the parameter of a procedure that capitalizes the characters of the string. Of course, this does not mean that the address and the label have the same structure.
2. There can be an error in the application (program, screen or report layout, etc.) or in the database (instance of the data that can lead to ignore an existing constraint or to find a spurious one). This happen quite often, because many of the difficulties enumerated for the data structure extraction are also present during the conception and the maintenance of the application. During the life time of the application some constraints can be added or removed and those modifications may be incompletely propagated to the entire program or data.
3. During the data structure extraction the analyst can also make some mistakes, i.e. misunderstanding some parts of the code, ignore some constraints, etc. These missing constraints can also lead to some conflicts.

The last two origins of conflicts are errors and will not be further discussed here, even if the probability to detect one of them (or both) in every project is very high (if not equal to one). Real applications are written by real programmers, they are maintained by real maintaineers and they are analyzed by real analysts with all these human factors and reliability that can be imagined. We have to keep in mind that information extracted from an existing system may be uncertain and incomplete.

A conflict (that is not an error but a misinterpretation) means that the physical construct is used to represent two (or more) different logical concepts and/or that some constraints are missing.

An example of a physical construct that represents two different logical concepts could be a file that contains two entity types with conflicting decompositions. This is a common practice in COBOL (especially in old programs) because the number of opened files is limited and this permit to open only one file where two can be necessary.

An example of a missing constraint would be a referential constraint between two entity types that is not always verified. This can mean that the referential constraint is optional and its presence or its absence depend of the value of another attribute.

4.8. Refinement methodology

The main problem of the data structure extraction phase is to discover and to make explicit, through the refinement process, the structures and constraints that were either implicitly implemented or merely discarded during the development process. The variety of implicit constructs can be very large. This chapter presents a generic refinement methodology.

Due to the large amount of information to manipulate attempting an exhaustive search for all the conceivable constraints is unrealistic. A methodology is needed that guides the analyst in his constraints investigation.

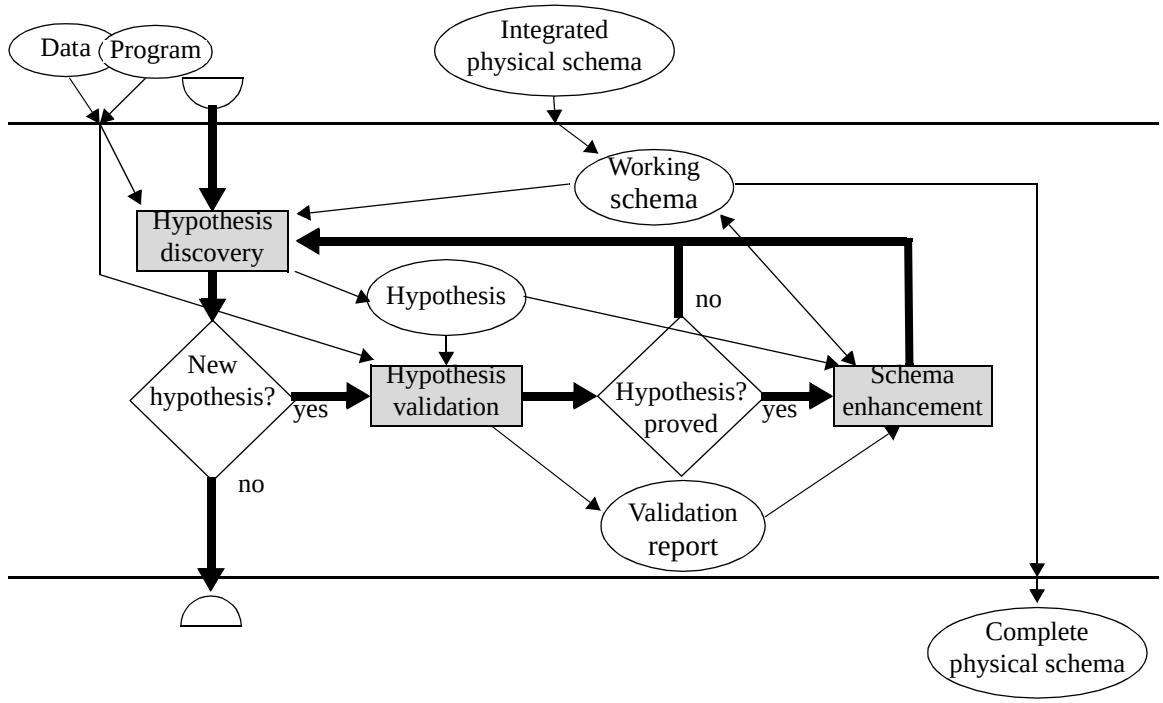


FIGURE 49. The refinement methodology.

4.8.1. The refinement methodology

The refinement methodology will reduce the search space to the possible constraints. For example, it is not realistic to query the data to check each attribute combination being an identifier. We have to decide which attributes are potential identifiers depending on their names (containing a keyword such as "code", "id", "num", etc.), their structure (mandatory), their position in the entity type (the first attribute of the entity type), etc.

Figure 49 sketches the proposed *Schema refinement* method. In this figure rectangles represent processes, ellipses represent the different products (schema, data passed form one process to the others) and diamond shapes represent decision points. The execution flow is materialized by bold arrows and plain arrows represent product usage. The schema refinement process receives the *integrated physical schema* and all the *information sources* such as input and produces the *complete physical schema* (see figure 29).

The schema refinement contains three processes:

- *Hypothesis discovery*

  This process analyzes the working schema and all the sources of information to find a potential missing construct or constraint, called *hypothesis*. Usually the hypothesis is discovered by the analysis of the name, structure of the physical schema or by some inexpensive analysis of the other sources of information. For example, a hypothesis can be "*Ref-A* of *B* is a referential attribute to the identifier of *A* (*Id-A)"*. The techniques used to discover this hypothesis can be name analysis (*Ref-A* contains the keyword "ref" and the name of the referenced entity type, *A*) or the structure analysis (length and type of *Ref-A* and *Id-A* are the same and *Id-A* is an identifier, etc.).

The purpose of this process is to guide the implicit constraint discovery. We focus our attention on some potential constraints using some inexpensive analysis techniques.

- *Hypothesis validation*

  The validation of the hypothesis discovered by the previous process is done by an in-depth analysis of the working schema and of some of the sources of information. This validation can require heavy and expensive analysis, such as program understanding or data analysis. For example, the analyst can use program understanding techniques to understand how the entity type *B*, specially the attribute *Ref-B*, is managed by the program. At the end of this process the analyst needs to be convinced that the hypothesis is valid or not.

- *Schema enhancement*

  Based on valid hypotheses (proved correct) and the validation report, the corresponding construct is added to the working schema. For example, in this example, the referential constraint from *Ref-A* to *Id-A* is created.

The schema refinement is iterated until no new *hypotheses* are generated by the *hypothesis discovery* process.

There is a pre and a post processing, not explicitly displayed in figure 49, that can be seen such as the input and output parameters definition. The pre-processing copies the integrated physical schema to obtain the working schema. Before the first iteration both schemas are the same. The post-processing copies the working schema to obtain the complete physical schema.

It is always possible to find a new hypothesis, but a limit must be set to the hypothesis discovery. One of those limits is that we are only interested by some kinds of constraints or constraints discovered by some analysis techniques. Another limit is to stop when the discovery of a new hypothesis or its validation become too expensive.

4.8.2. Hypothesis validation

A good practice is to apply as many heuristics as possible to validate (or disprove) a hypothesis. If a heuristic succeeds that does not mean that the hypothesis is verified. For example, data analysis is used to validate the hypothesis "*A1* is an identifier of *A*" and this heuristics is applied on a small set of data in which, by chance, all the value of *A1* are unique. If the test is done again on a larger data sample some non unique value of *A1* may be found. On the opposite, if a heuristic fails, the hypothesis is not necessarily disproved. For example, during the verification the hypothesis that there is a referential constraint between two entity types and a source code, a fragment of code is found where the referential constraint is not checked before the entity type is written. This does not mean that the referential constraint does not exist, it can be a mistake in the program, the programmer forgot to validate the referential constraint in this fragment of the code. Or this referential constraint is an environment constraint and thus it is not validated by the program. Or it is an optional referential constraint and the analyzed code fragment writes the entity type instances for which the referential constraint is not present. This can be formalized as follows:

- Let *h* be an hypothesis on the existence of an implicit construct *C* (for example, attribute *B2* of *B* is a referential attribute to the identifier of *A*, *A1*)*; so* far, *h* is stated with probability $p_0 < 1$[1].

1. The probability (p_0 and p_1) do not have a numerical value but represent the trust the analyst has.

- The heuristics *H* is applied; *h* is now stated with probability p_1:
  - if *H* succeeds
    $p_1 > p_0$; the existence of *C* is more certain, though $p_1 < 1$.
    For instance, in the example above, if there is an index on *B2* it is one more evidence that *B2* is a foreign key to the identifier *A1* of *A*, but we are not yet completely certain.
  - if *H* fails, one the three interpretations can hold:
    - $p_1 = 0$; *h* is disproved, the constraint *C* does not exist.
      For example, half of the values of *B.B2* are not in *A.A1* value set, thus there is no referential constraint from *B.B2* to *A.A1*.
    - $p_1 < p_0$; *h* is less certain, but could still be proved through other heuristics.
      For example, there is only one value of *B.B2* (out of one million) that is not in *A.A1* value set. It cannot be concluded that there is no referential constraint, it is perhaps an error in the data.
    - *H* does not contribute to the search; $p_1 = p_0$.
      For example, there is no data stored in *B* so no data violated the constraint and no data that validated it!

Experience has exhibited some restrictions in the application of this method in real projects. Analyzing all the information sources with all the heuristics generally proves too expensive, so that the analyst has to determine which sources to analyze with which heuristics. There are no general rules to decide which source of information to use, the analyst has to choose depending on what he is looking for, what are the respective quality of the different sources of information, the tools available, etc.

This method considers that all the information sources are reliable. What about the result of a heuristic applied on unreliable information (corrupted data, programming errors, etc.)? This is why it is suggested to apply more than one heuristics and to analyze more than one source of information because in real projects some unreliable information can be found. A hypothesis cannot be proved by heuristics alone, it is up to the analyst to decide when he is convinced that the hypothesis is validated or invalidated. Hence the importance of the analyst's skill and his knowledge of the application domain.

During a hypothesis validation other constraints can be discovered, these constraints must be added to the schema (the opportunistic approach [Tilley-1998]). For example, to validate a referential constraint, the analyst analyzes a program slice. This slice contains, in additions to the validation of the referential constraint, the instructions that verify that the value of the referential constraint is unique (the attribute is also an identifier).

This method is presented as an algorithm, but it is far from being deterministic. The *hypothesis discovery* and *hypothesis validation* processes heavily rely on the analyst skills and on the tools he uses.

4.8.3. How to decide that refinement is completed

The ending condition is one of the most difficult and less formalized point of the refinement methodology. It is impossible to know when all the implicit constraints have been discovered, because there is no reference schema with which the current schema can be compared.

The analyst is never sure that he has found all the constraints. There is always some part of the code that has not been analyzed or some heuristics that have not been applied.

So the analyst has to decide, with some methodological and economical guideline, that the refinement process is terminated. An important element to decide if the refinement process is finished, is to know before to start the DBRE project why the DBRE is done and what are the expected results. The simplest DBRE project is to recover only the list of all the entity types with their attributes as declared in the DDL. In such a project the refinement process is not done at all, the ending condition is always true. The result of such projects is the integrated physical schema. The final product is an incomplete logical schema. This can be useful to make a first inventory of the data structures used by an application to prepare another reverse engineering or maintenance project.

On the other hand, all the possible constraints may have to be recovered to get a complete view of the database. This is necessary in migration or maintenance projects. For example, to add new functionalities to an existing application, all the constraints of the database need to be known before any modification. Otherwise, there is a risk to add some functions that corrupt the data.

Due to time and budget limitation, the analyst has to decide, in dialogue with the customer, what kind of constraints he is looking for and which heuristics he is using. The customer has to explain to the analyst what he wants to do with the DBRE results, the time and budget he can devote to this project. Then the analyst can say if it is possible to perform the project within the given time and budget. The analyst has also to explain the analysis techniques he will use and what are the weakness and strengths of the proposed solutions.

4.8.4. Refinement strategy

To have a homogeneous result, it is preferable to search for constraints using heuristics on the whole information source than to apply a lot of heuristics but only on a part of this source of information. For example, it is preferable to use dataflow analysis on the complete source code, than to apply dataflow analysis, program slicing, etc. on only some part of the source code. Indeed, if the analyst applies a lot of heuristics on a subset of the information source that cover only a part of the schema, the resulting schema will not be homogeneous. For some part of the schema a lot of implicit constraints will be recovered while on another one very few (or none) of the implicit constraints will be recovered. It is particularly difficult to use such a schema when a constraint is not present. The analyst is never sure that the constraint is not present because this part of the schema was not analyzed or because the constraint does not exist.

For example, when the analyst uses dataflow analysis to recover referential constraint. He applies this techniques on the source code of the application. If there is an entity type that is not connected to the rest of the schema, he can be confident that this entity type is not connected with another entity type of the schema. But if he has chosen to apply dataflow analysis to only a part of the source code of the application and there is an entity type that is not connected to the rest of the schema. Does it mean that this entity type is not connected to another entity type or that the code that materialize the connection with another entity type was not analyzed?

	dataflow analysis	clichés analysis	program slicing	name analysis	physical structure	screen/report	data analysis	program execution	domain knowledge
fine-grained structure	DV		DV		D	DV	V		D
optional fields			V			DV	V	DV	D
field aggregates	DV		DV	D		DV	V	DV	D
multivalued fields	DV		DV	D		D	V	DV	D
multi structure	DV		DV		D		V		D
record identifiers		V	V	D	D	DV	V	DV	D
field identifiers		V	V	D		DV	V	DV	D
foreign keys	DV	V	DV	D	D	DV	V	DV	D
functional dependencies	DV	V	DV	D		DV	V	DV	D
array / set		V	V		D	DV	V	DV	D
existence constraints		V	V	D		DV	v	DV	D
exact cardinality		V	V			DV	V	DV	D
enumerate domains	DV	V	DV			DV	V	DV	D
const. on domain		V	V			DV	V	DV	D
meaningful name	DV		DV	D		DV		DV	D

FIGURE 50. The techniques applied to discover constraint (D = hypothesis discovery, V = hypothesis validation).

4.8.5. Heuristics usage

In figure 50, we give, for each implicit constraint, the elicitation techniques that can be used to discover it. We note if the technique is better suited for hypothesis discovery (D) or hypothesis validation (V). This table is not exhaustive (rigid). As usual it is possible to find some project where hypothesis discovery or hypothesis validation can be performed by an heuristic that we have not been noted. This table is the result of our experience and the analysis of the capabilities of the different elicitation techniques.

For example, it is possible to imagine a program where all the optional attribute's names are suffixed by the keyword 'opt'. So name analysis can be used to discover the optional attribute. But our experience shows us that this is improbable.

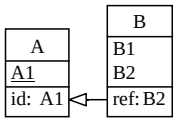


FIGURE 51. The concept of foreign key.

4.8.6. Application to foreign key elicitation

This section illustrate how the refinement methodology is applied to the referential constraint discovery.

A referential constraint is an attribute (or combination thereof) of each value that is used to reference an entity type. The standard configuration of referential constraint *B2* can be symbolized by *B.B2 >> A.A1*, where *B2* is a single-valued attribute (or a set of attributes) of entity type *B* and *A1* is the primary identifier of entity type *A*. *B.B2* and *A.A1* are defined on the same domain (figure 51). However, practical referential constraint do not always obey the strict recommendations of the relational theory and richer patterns can be found in actual applications [Hainaut-1997a].

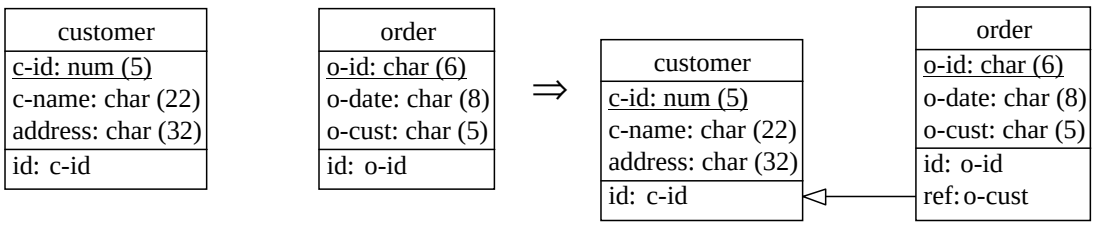


FIGURE 52. Foreign key elicitation, the source and final schema.

Let us base the discussion on the schema of figure 52, in which two entity types `customer` and `order` may be linked by a referential constraint. It is assumed that *c-id* is the identifier of *customer* and there should exist a referential constraint in *order* that would reference this identifier (in short, the target identifier is known).

The refinement methodology is applied to find the referential constraint that exists between *order* and *customer*. The remainder of the section is divided into two parts. The first one presents the different techniques that can be applied to discover hypotheses. The second part applies the different elicitation techniques to the hypothesis validation.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT CUSTOMER ASSIGN TO "CUST.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS C-ID.
  SELECT ORDER ASSIGN TO "ORDER.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS O-ID
    ALTERNATE RECORD KEY IS O-CUST
                WITH DUPLICATES.
DATA DIVISION.
FILE SECTION.
  FD F-CUSTOMER.
    01 CUSTOMER.
      02 C-ID          PIC 9(5).
      02 C-NAME        PIC X(22).
      02 C-ADDRESS     PIC X(32).
  FD F-ORDER.
    01 ORDER.
      02 O-ID          PIC X(6).
      02 O-DATE        PIC 9(8).
* O-CUST REFERENCE CUSTOMER
      02 O-CUST        PIC X(5).
WORKING-STORAGE SECTION.
  01 C                   PIC 9(5).
```

```
PROCEDURE DIVISION.

PRINT-REPORT SECTION.
ASK-CUST.
  DISPLAY "ENTER CUSTOMER NUMBER ".
  ACCEPT C.
  MOVE C TO C-ID.
  READ F-CUSTOMER
    INVALID KEY GO TO  ERROR-1.
  MOVE C TO O-CUST.
  MOVE 1 TO END-CUST.
  PERFORM DISP-ALL-ORD UNTIL END-CUST=0

DISP-ALL-ORD.
  READ F-ORDER
    INVALID KEY MOVE 0 TO END-CUST.
  IF END-CUST = 1
    PERFORM DISPLAY-ORDER.


CREATE-ORD SECTION.
  DISPLAY "ENTER CUSTOMER NUMBER".
  ACCEPT C-ID.
  READ F-CUSTOMER
    INVALID KEY GO TO  ERROR-2.
  MOVE C-ID TO O-CUST.
*ASKS FOR THE ORDER INFORMATION
  ...
  WRITE ORDER
    INVALID KEY GO TO ERROR-3.
```

FIGURE 53. Excerpts from a program working on `customer` and `order`.

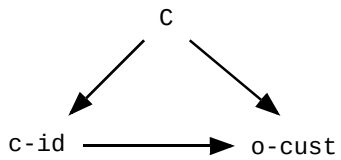


FIGURE 54. The dataflow graph of the code of figure 53.

4.8.6.1. Hypothesis discovery

A. Dataflow analysis

If a referential constraint holds between two entity types, then it should exist, in some program, a dataflow between variables that represent the referential constraint and the target identifier. Considering assignments and equality relations in the code of figure 53, the dataflow graph of figure 54 is computed. It shows that, at given time points, *c-id* and *o-cust* share the same value. It is reasonable to think that this property is always verified, hence the hypothesis that attribute *o-cust* could be a referential attribute is confirmed.

```
Customer : Smith
           Order      Date
           1          1-1-200
           5          1-10-2000
           9          2-3-2001
Customer : Dupont
           Order      Date
           2          3-2-2000
           6          5-11-2000
```

FIGURE 55. Example of a report that display customer and order.

B. Screens / forms / reports layout analysis

Reports can be considered a populated view of the persistent data. The analysis of the report of figure 55 shows that below each customer there is a list of orders. This analysis suggests that there is a referential constraint from `order` to `customer`.

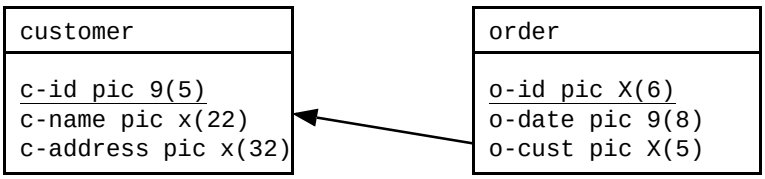


FIGURE 56. An fragment of the application documentation.

C. Current documentation analysis

The code of figure 53 includes comments that suggest that `o-cust` is a referential constraint that references *customer.*

There still exists some documentation written during the development of the application. Figure 56 shows an excerpt of the documentation of the application that describes *o-cust* as the referential attribute that references *c-id*.

D. Domain knowledge usage

Everybody knows that customers place orders. Obviously, entity types *customer* and *order* should be linked in some way.

E. Program execution analysis

The program refuses to delete a *customer* record because the *customer* still has pending *order.* This behavior can be translated into the fact that *order* entity type depends on the *customer* entity type.

F. Physical structure analysis

A referential constraint is a mechanism that implements a link between entity types and is the privileged way to represent inter-entity relationships. We can assume with little risk that application

programs will navigate on records following these relationships. Therefore, most referential constraints will be supported by such access mechanisms as access key. *Heuristics*: an attribute supported by an access key could be a referential attribute, specially when it is not an identifier. In the example an access key (`record key with duplicate`) has been declared on `o-cust`.

Quite naturally, the candidate attribute should have the same domain of values, i.e. the same type and length, as the identifier. However, some matching distortions can be found as far as lengths and even types are concerned. *Heuristics*: the candidate referential attribute must match, strictly or loosely, the identifier of the candidate referenced entity type. In our example, `o-cust` is declared as `pic X(5)` and `c-id` is `pic 9(5)`, the lengths are the same but the types are not. This is a common COBOL habit to mix alphanumeric characters and numeric characters.

G. Name analysis

The name *o-cust* includes a significant part of the name of the target entity type (*customer*). This suggests that *o-cust* references *customer.*

4.8.6.2. Hypothesis validation

During the hypothesis discovery process a potential referential constraint has been discovered. Depending on the heuristic used the hypothesis can be formulated in two different ways:

- There is a referential constraint between *CUSTOMER* and *ORDER.*
- There is a referential constraint between *CUSTOMER.O-CUST* and *ORDER.C-ID.*

Usually, hypothesis validation heuristics are techniques that give reliable result but are more expensive to apply.

A. Cliché analysis

To validate the referential constraint hypothesis, each `WRITE` (and `REWRITE`) instruction that stores the origin entity type must match a referential constraint cliché. Indeed to enforce a constraint the program must verifies it before storing the entity type.

There is half a dozen clichés which make it possible to detect a referential constraint in a COBOL program. Cliché of figure 45 on page 78 matches with the `CREATE-ORD` section of the code of figure 53.

B. Program slicing

To validate a referential constraint hypothesis, the analyst must check that each time the *ORDER* entity type is stored there is a fragment of code that ensures that the referential constraint is verified. This fragment of code must be executed before a `WRITE` (or `REWRITE`) instruction and it must contain an access (`READ`) to the target entity type. Program slicing can be used to discover such a fragment of code. The program slice computed with respect to a `WRITE` (or `REWRITE`) instruction of the referential constraint entity type must contain a `READ` instruction to the target entity type. This slice is analyzed to detect which attributes (in the origin entity type and in the target entity type) are

used to ensure the referential constraint. Such a program slice must be computed and analyzed for each WRITE (or REWRITE) instruction to ensure that the referential constraint is always verified.

```
CREATE-ORD SECTION.
  DISPLAY "ENTER CUSTOMER NUMBER".
  ACCEPT C-ID.
  READ F-CUSTOMER
    INVALID KEY GO TO  ERROR-2.
  MOVE C-ID TO O-CUST.
*ASKS FOR THE ORDER INFORMATION
  ...
  WRITE ORDER
    INVALID KEY GO TO ERROR-3.
```

FIGURE 57. Program slice of program of figure 54 with respect to WRITE ORDER.

In the example of figure 53, the program slice is computed with respect to the WRITE ORDER instruction (see figure 57). The analysis of this slice proves that before ORDER storage, the program checks if O-CUST is an existing value of C-ID (the identifier of CUSTOMER).

```
MOVE 0 TO NUM-ERR.
MOVE 0 TO NUM-ORD.
MOVE 1 TO END-FILE.
PERFORM READ-ORD UNTIL END-FILE = 0.
DISPLAY "number of order: " NUM-ORD.
DISPLAY "number of error: " NUM-ERR.
```

```
READ-ORD.
  READ F-ORDER NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      PERFORM VERIF-CUST.

VERIF-CUST.
  ADD 1 TO NUM-ORD.
  MOVE O-CUST TO C-ID.
  READ F-CUSTOMER
    INVALID KEY ADD 1 TO NUM-ERR.
```

FIGURE 58. Program that counts the number of O-CUST that does not appear in C-ID.

C. Data analysis

To validate a referential constraint through the data analysis, the database contents needs to be queried to know the number of instances that violate the constraint. This can be easily done with modern DMS as SQL. For such DMS it can be implemented as a query. For less powerful DMS (as COBOL) a small program needs to be written to query the database.

The program of figure 58 counts the number of *ORDER* that violate the referential constraint. This sample program does not only count the number of erroneous *ORDER* but also the total number of *ORDER*. In the idealistic situation, where the data does not contain errors, the among of erroneous data is enough to decide if the constraint is present or not. If the number of errors is 0 then the referential constraint is validated otherwise there is no referential constraint. On the other hand, in real databases there are often errors in the data and thus it is useful to know the number of errors with respect to the total number of instances. If the number of errors is very small with respect to the total number of data, the analysis can assume that the referential constraint is verified.

D. Program execution

If the program rejects any tentative data entry concerning an `order` unless its `o-cust` value appears as the `c-id` value of some `customer` record, then we can conclude that the program enforces some kind of inclusion property between these value sets, which can be interpreted as referential integrity.

CHAPTER 5 *Program understanding in database reverse engineering*

At first sight, it can be strange to use program understanding to perform DBRE. DBRE recovers the data structure that is more or less independent from the programs. It is shown in this chapter that the implicit constraints can be elicited through the analysis of the source code. The source code is one of the most accurate and up-to-date sources of information for the recovery of the implicit constraints. Due to the difficulty and expensiveness of source analysis, the analyst must have program understanding techniques and tools .

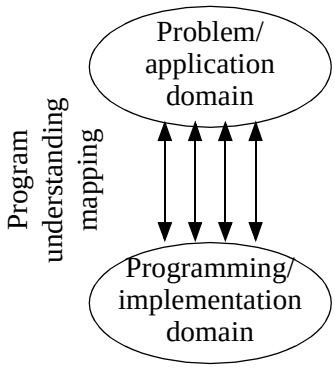


FIGURE 59. Program understanding is the mapping between the program and the problem domain.

5.1. Program understanding

Program understanding is the process of acquiring knowledge about a computer program [Corbi-1989][Rugaber-1995][Tilley-1998b][von Mayrhauser et al.-1994][Young-1996]. Increased knowledge enables such activities as reverse engineering, documentation, bug correction, enhancement and reuse. Program understanding is not only understanding the code, but also the mapping between program domain and problem domain (see figure 59).

While efforts are underway to automate the understanding process, such significant amounts of knowledge and analytical power is required that today program understanding is largely a manual task.

In [Rugaber-1995], Rugaber explains that program understanding is difficult because it must bridge different conceptual areas. He describes five gaps that need to be bridged:

1. *Application domain and program*

   Programs are solutions to problem situations from some application domain. It is the job of the person trying to understand the program to reconstruct the mappings from the application domain to the program.

2. *Physical machines and abstract descriptions*

   Computer programs are incredibly detailed. One of the jobs of the analyst is to decide, from all the programs details, which are the important concepts (abstraction).

3. *Coherent models and incoherent artifacts*

   During the design the program is constructed as a coherent set of details. Through maintenance activities such as porting, bug fixing and enhancement, the original structure of the program may have deteriorated. The analyst needs to detect the high level structure of the program when the original purpose of the program may have change.

4. *Hierarchical world of program and associative nature of human cognition*

   Computer programs are highly formal. Human cognition seems to work associatively. The process of human understanding is controlled by expectations derived from the application domain and the programming knowledge. A program is understood to the extent that the reverse engineer can build up correct high level chunks from the low level details evident in the program.

5. *Bottom-up program analysis and top-down synthesis*

   When an experienced analyst looks at a program, he detects patterns that indicate the intent of some section of code (bottom-up). At the same time, he has some idea of the overall purpose of the program and how it might be accomplished (top-down). The difficulty is that both of these activities need to proceed at the same time, in a synchronized fashion.

Most program understanding work is currently done by humans. To understand a program three kinds of actions can be taken [Corbi-1989]:

- Read about it: the analyst can read (or analyze) any kind of documentation available about the program. The problem is that the documentation does not always exist and when it exists it is not necessarily up-to-date, correct or well written.
- Read it: the analyst can read (or analyze) the code itself. Usually the code is the primary source of information because it is the only really accurate representation of the system.
- Run it: an interesting source of information about a program is its execution to analyze (through some tracing processes) how it work on real data.

In order to study the program understanding process, it is important to look at the human factors involved in comprehension. The *cognitive aspect* [Corbi-1989] [Tilley-1998b] of program understanding is the study of the problem-solving behavior of software engineers engaged in understanding tasks. Three theories appear in the literature: the bottom-up, the top-down and the opportunistic one. In the *bottom-up* theory, by reading the code, an analyst essentially iteratively abstracts a higher-level understanding of the program by recognizing and then naming more and more of the program. The *top-down* theory proposes that programmers use their own experience and repeatedly

try to confirm their expectations or the basic of what they believe the design to be. Now, when they pick up the code, they look for where these elements occur and fill in their belief of what the design most probably is. If something is missing or radically different from his expectations, the surprise causes some new experience to be stored for the next encounter. The *opportunistic* theory says that understanding is a mixture of top-down and bottom-up strategies. Understanding a program involves a knowledge base (which represents the expertise and background knowledge of the analyst), a mental model (which is an encoding of the analyst's current understanding of the program) and an assimilation process.

Because the productivity of software engineers varies by more than an order of magnitude, the strategies of successful practitioners are of great interest in producing methodologies, tools and techniques that better support program understanding. These tools proceed from straightforward textual analysis to the dynamic analysis of executing programs. The main program analysis techniques are the following:

1. *Textual analysis*

   One of the simplest ways to understand a program is to manually flip through source code or to search for a given string.

2. *Syntactic analysis*

   The syntactic analysis is performed by a parser that decomposes a program into expressions and statements. The result of the parser is stored in a structure called an *abstract syntax tree* (AST) that is the basic of most sophisticated program analysis tools. Because an AST is a tree, it can be traversed or queried.

3. *Control flow analysis*

   There are two forms of control flow analysis. Intraprocedural analysis provides a determinator of the order in which statements can be executed (sequence, condition, loop, etc.) within a program. Interprocedural analysis determines the calling relationship among the program unit, as call graph.

4. *Dataflow analysis*

   The dataflow analysis is the analysis of the flow of the values from variables to variables between the instructions of a program. To compute the dataflow of a program the variables defined and the variables referenced by each instruction must be known. A variable is defined by an instruction, if the instruction modifies the value of the variable (e.g. through an assignment). A variable is referenced by an instruction when its value is used by the instruction (e.g. a variable that appears in a conditional instruction). Dataflow analysis is concerned with answering questions related to how definition flows are used in a program.

5. *Slicing*

   The slice of a program with respect to program point *p* and variable *v* consists of all the program statements and predicates that might affect the value *v* at point *p*. This concept was originally discussed by M. Weiser [Weiser-1984], see 6.4.

6. *Cliché recognition*

   A cliché is a programming pattern. The program source code can be searched for these pattern. An example of a cliché is a pattern describing loops for performing linear search.

7. *Abstract interpretation*

   The basic idea behind abstract interpretation is to approximate (usually undecidable) properties by using an abstract domain instead of the actual domain of computation. As a consequence, the program as a whole can be given an approximated meaning, hopefully capturing interesting

properties while leaving out irrelevant detail as much as possible.

8. *Dynamic analysis*

   The analysis techniques described so far have all been static that is they are performed on the source code of the program. It is also possible to gain increased understanding by systematically executing a program. This process is called dynamic analysis.

Such support mechanisms can manage the complexities of program understanding by helping the analyst extract high-level information from low-level code. These support mechanisms free analysts from tedious, manual and error-prone tasks such as code reading, searching and pattern-matching by inspection.

5.2. Program understanding in database reverse engineering

At first glance, it can be strange to use program understanding to perform DBRE. In the introduction of this thesis, we have explained that we restrict ourselves to the DBRE because it can be performed more easily than the reverse engineering of the procedural part of the application and because the database is independent from the application. Moreover the understanding of the underlying database can ease the understanding of the whole program.

Some authors only use the DDL, the physical schema or data themselves to reverse engineer the database. This approach can be valid if the DMS is powerful enough to express all the constraints and the programmer has used all the expressiveness of the DMS when he has developed the application. In such situations all the constraints are explicitly declared and there is no implicit constraint. Such condition can be verified in some modern and well designed (academic?) databases. But it can not be assumed in general and certainly not for legacy system.

Legacy DMS do not offer a rich set of constraints and the programmer needs to express complex constraints as referential constraints, data dependency, multivalued attributes, etc. Thus the programmer implement these constraints as implicit constraints (non declarative structure). As said in the previous chapters, even constraints or data structures that can be explicitly declared in the DMS are not always declared and are implicitly implemented (structure hiding). Constraints are not explicitly declared in the DMS for numerous reasons: reusability, genericity, simplicity, efficiency, poor programming practice, previous version of the DMS does not support such constraints, disorganization that results from prolonged maintenance [Tilley-1998b].

All the implicit constraints are recovered during the data structure extraction and more precisely during the schema refinement process. Without the recovery of the implicit constraints DBRE only produces another (graphical) view of the physical schema. The DBRE takes all its significance if it enhances the semantic of the physical schema to produce the conceptual schema.

All the DBRE processes, except the schema refinement, are quite well known. The DDL analysis is studied since the early 80's (see 1.4) and there exist many commercial tools to perform it. The data structure conceptualization is taught in schools and universities and there exist commercial tools to support it. But there exists very little research to tackle the schema refinement. The only implicit constraint discovery techniques usually suggested by some methodologies is the analyst domain

knowledge or some knowledge about the program. But they do not suggest how this knowledge is acquired!

In real projects also most of the refinement process relies on the analyst's knowledge and on a lot of manual work.

The schema refinement process gives very important added values to the logical schema. Since 1992, some authors have recognized that the procedural part of the application programs is an essential source of information to retrieve data structures ([Anderson-1996], [Hainaut et al.-1993a], [Joris et al.-1992] and [Petit-1996]) and that understanding some programs aspects is one of the keys to fully understand the data structure. In data-oriented applications, many (if not all) data structures and constraints that are not explicitly declared are coded, among others, as a procedural section of the program. The code is the only really accurate representation of the system.

The data in a database are the result of the executions of the programs which update the database. Therefore all the implicit constraints can be inferred from the ways that the programs update the database. If the database satisfies some constraints before the execution of a program and still satisfies these constraints after the execution, then the program must verify that the modified data do not violate the constraints. Thus a program must validate all the constraints before modifying the data. Some constraints can also be found through the analysis of programs that only access the data and do not modify them. Indeed when a program reads data, to print a report or display results on a screen, it uses some of the database constraints. For example, if there exists a referential constraint between the *CUSTOMER* entity type and the *ORDER* entity type, this referential constraint is used, to access to find the customer when the invoice is printed.

The program source code is an accurate and up-to-date source of information. Programs are the only way for the users to access the data so all the constraints that the programmer wants to enforce must be present in the code. Programming languages are very precise and deterministic, there is only one interpretation of what a fragment of program is doing. The knowledge acquired through the analysis of the code is quite sure. The source code is an up-to-date source of information because it is used to generate (compile) the application.

The drawback of source code analysis is that it is a difficult and expensive task. The difficulty comes from the fact that the programs are written using legacy languages that the analyst needs to master. The analyst needs to have a deep understanding of the language to understand programs that have been written by other programmers. The analyst must also have a good knowledge of the forward engineering process to understand the code produced by other programmers. The size of the application is also a source of difficulties. It is not rare to have application of several 100000 LOC.

For all those reasons, the program text source is a very useful source of information in which we can discover a lot of implicit construct during the data structures extraction. But to use this source of information effectively, the analyst needs program understanding techniques and tools.

Another asset of procedural code analysis is that it can help in the comprehension of the data's semantics. The understanding of the business rules, the data manipulation algorithms give important hints to understand the meaning of the data and thus to increase the domain knowledge of the analyst.

5.3. Program understanding difficulties

The analysis of the program source is a complex and tedious task. This is due to the fact that procedurally-coded data constructs are spread in a huge amount of source files, can be duplicated, and also because there is no standard way to code a definite structure or constraint.

As an example of this, there is only one way to declare explicitly a referential constraint in SQL-DDL (...`foreign key <column> reference <table>`), it is done only once (in the database declaration) and the constraint is always satisfied. This declaration is easy to detect in the DDL code. Once it has been found, the analyst is sure that the constraint is present and he can add it to the schema without any other verification.

```
new-order1.
  display "customer number".
  accept CUS-ID.
  read CUS key CUS-ID
    invalid key go to error.
  move CUS-ID to ORD-CUS.
  display "order number".
  accept ORD-ID.
  ....
  write ORDER.
```

```
new-order2.
  display "customer number".
  accept ORD-CUS.
  move ORD-CUS to CUS-ID.
  read CUS key CUS-ID
    invalid key go to error.
  display "order number".
  accept ORD-ID.
  ....
  write ORDER.
```

```
new-order3.
  display "customer number".
  accept WO-ORD-CUS.
  move WO-ORD-CUS to CUS-ID.
  read CUS key CUS-ID
    invalid key go to error.
  move WO-ORD-CUS to ORD-CUS.
  display "order number".
  accept ORD-ID.
  ....
  write ORDER.
```

```
new-order4.
  display "customer number".
  accept WO-ORD-CUS.
  move WO-ORD-CUS to CUS-ID.
  read CUS key CUS-ID
    invalid key move 0 to FIND-CUS
    not invalid key move 1 to FIND-CUS.
  if FIND-CUS
    move WO-ORD-CUS to ORD-CUS
    display "order number"
    accept ORD-ID
    ....
    write ORDER
  end-if.
```

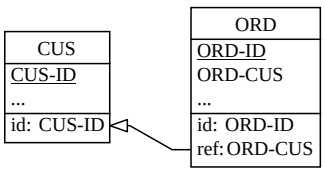


FIGURE 60. Four different fragments of code that verify the same referential constraint.

On the other hand, there are many different ways to implement an implicit referential constraint. Each time an occurrence of one of the entity types that take place in the constraint is modified, inserted or deleted, code that validates the constraint must be produced. The code is scattered in the

application and can be different. To be sure that the constraint is always verified, the analyst needs to check all the fragments of code that modify, insert or delete the entity types.

Figure 60 presents four fragments of COBOL code that verify the same referential constraint (between `ORDER` and `CUSTOMER`) before the recording of a new `ORDER` (implicit constraint), and we can easily imagine other algorithms to validate this referential constraint that uses `go to`. From the user point of view, those four fragments verify exactly the same constraint. The pattern (or cliché) to search for the discovery of the referential constraint is different in each example. In the first one the dataflow is from `CUS-ID` to `ORD-CUS`, in the second one it is the opposite. In the third one an intermediate variable (`WO-ORD-CUS`) is used and the dataflow is from this variable to `CUS-ID` and to `ORD-CUS`. This shows the different usage patterns that must be checked to detect very similar constraint; there are many other ways to verify a referential constraint in a source code. Those examples are simple because they are entirely written in COBOL and these code fragments are adjacent lines of code that only validate the referential constraint. It is easy to imagine that the complexity can increase if the instructions are spread over different paragraphs that perform many other verifications. So the analyst has to gather the pertinent instructions by the analysis of thousands of lines of codes, following the control flow (`if`, `go to`, `perform`). This complexity can also be increased by the use of some embedded DMS queries (as SQL). The analyst (and the tool) needs to understand (parse) two different languages. To understand programs with embedded DMS queries, another difficulty is that the DMS physical schema is not declared in the programs. The physical schema must be extracted from the DMS-DDL and the analyst must do the mapping between the physical schema and the program's variables.

Each programmer has his personal way(s) to express the constraints (variable naming, comments usage, algorithm, code presentation) depending on his skill, his programming experience, his mood. This will increase the understanding difficulty because the analyst has to first discover how the programmer has worked. For example, if the programmer uses a variable (`cus-name`) to store two different information (the customer name and the product name) to save memory space, it can be very difficult to read (understand) the code. The constraints coding also depends on the programming language, the target DMS, the enterprise rules (naming convention, comments usage, features of the DMS used), the level of optimization needed, the history of the application (the maintenance process, the different migrations), the tools used. For example, if the name of the origin of a referential attribute contains the name of the target attribute or entity type, it can ease this referential constraint discovery. But if both attributes have meaningless names, it can be very difficult to understand the referential constraint validation code.

The same section of code can be used to validate several constraints. For example, if the *ORDER* entity type contains a multivalued attribute to store the products ordered and if a product can only be ordered once per order. The section of code, that validates a new product added to the order, must check that the new product is a valid product and that the product was not already ordered in the current order. The analyst needs to dissociate the different constraint validations.

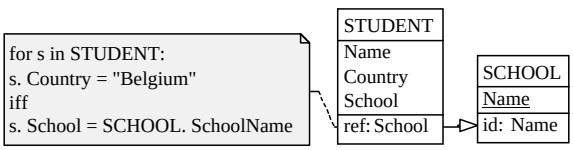


FIGURE 61. Example of optional referential constraint.

If the constraints are not explicitly declared in the database, each module (or function) that modifies the data must verify them. Thus the validation code is duplicated and each occurrence of the validation can be coded differently. Quite often at least one of the validation code does not verifies the same constraints as the other. This can have different interpretations. It is possible that the analyst has misunderstood the constraint and must find another interpretation that includes all the code fragment. For example, if a code fragment validates the referential constraint shown in figure 61 and another one does not validate it. It can also be interpreted as the fact that the referential constraint is optional, it is only present if the *STUDENT.Country* attribute is equal to "Belgium", otherwise *STUDENT.School* does not reference *SCHOOL.Name*. Another reason for which two fragments of code do not implement the same constraint, is that there are errors in one of them.

5.4. Program understanding techniques in DBRE

We do not need to retrieve the complete program specification; we are merely looking for evidence that are relevant to find the undeclared structures and constraints on persistent data. More precisely, we are looking for evidence of the implicit structures and constraints described in chapter 4 as attributes refinement and aggregation, referential constraints and exact cardinalities, to mention only a few.

Several industrial projects (see 9.4) have proved that powerful program understanding techniques and tools are essential to support data structure extraction process in realistic size DBRE projects.

We have studied and adopted some program understanding techniques that ease the discovery of those implicit constraints. These program understanding techniques are:

- *Pattern matching*: the search for patterns in a source text.
- *Variables dependency graph*: this is a graph where the nodes represent variables of the program and the arcs are relations (usually dataflow) between the variables.
- *Program slicing*: extract from a program only the lines necessary and sufficient to understand the value of a variable at a given instruction.
- *Program visualization*: different representations (call graph, data flow diagram,...) of a program.

All those techniques will be studied in detail in the following chapter.

CHAPTER 6 *Program understanding techniques*

Program understanding techniques can be adapted to retrieve the implicit constraints on data that are implemented in the programs. This chapter presents five of those techniques used to understand the mentioned data constraints: pattern matching, variable dependency graph, program slicing, system dependency graph and graphical visualization of programs.

6.1. Introduction

As said in the previous chapter, program understanding has been developed by the software engineering communities to acquire knowledge about programs for debugging, maintenance, enhancement and reuse. Many techniques have been developed to help the analyst in the comprehension of the existing programs.

This thesis focuses on the program understanding used in database reverse engineering. To perform this task, the analyst does not need to understand all the aspects of the program but wants to map the persistent data usage to a database schema. This chapter presents different program understanding techniques that can be used to better understand how the persistent data are used and which constraints, on those data, are ensured by the program. Different program understanding techniques, that can be used to better understand how the persistent data are manipulated, are studied:

- *Pattern matching*

  Search of patterns (strings) in source code.
- *Variable dependency graph*

  A graph that represents the relation between the variables.
- *Program slicing*

  Decomposition technique that extracts from program statements relevant to a particular computation. An extension of program slicing to analyze programs with embedded SQL is also presented.

- *System dependency graph analysis*

  The system dependency graph is the program representation used by program slicing. It is possible to imagine other usage (querying) of the system dependency graph than to compute slices.
- *Graphical visualization of program*

  Some aspect of the program architecture as call graph, data usage, can be visualized as graphs.

6.2. Pattern matching

The simplest program understanding technique is to search for a string in the text sources. The technique presented to search for a string in a text source is not a simple string searching tool, but a more sophisticated pattern matching engine. The term pattern is used and not just string, as in a text editor, because a pattern describes a set of possible strings. It can include wildcard, characters ranges, multiple structures and variables and can be based on other defined patterns. For example, a simplified version of the COBOL assignment (`Move A to B`) can be defined as the characters "`MOVE`", followed by at least one separator (space, new line, tab,...), followed a COBOL variable (that must be defined before), followed by at least one separator, followed by the characters "`TO`", followed by at least one separator, followed by a second COBOL variable.

We have defined a *Pattern Definition Language* (PDL) has been defined to describe the patterns. This language is close to the BNF notation; it defines the following structures (the complete PDL syntax can be found in the annex A.1.1):

- *Terminal segment*

  A string that is matched as it is, the matching is case sensitive or not.
- *PDL variable*

  This is a variable name that is assigned to an already defined pattern. If a variable with the same name appears more than once in a pattern, then each occurrence of the variable must have the same value. A value can be assigned to the variable, before the search takes place, to specialize the pattern. The value that matches to the variable can be used by other processors.
- *Range of characters*

  Matches any character belonging to the range.
- *Optional segment*

  This segment can be matched to the empty string.
- *Repetitive segment*

  This segment can appear more than once.
- *Choice segment*

  It must match one of the segments of the choice.
- *Regular expression*

  Pattern can contain grep regular expression ([Robbins-2002]).
- *Pattern*

  A pattern definition can contain a reference to an already defined pattern.

The COBOL assignment can be expressed in PDL as follow:

```
move ::= "MOVE" - @var_1 - "TO" - @var_2;
```

where `move` is the name of the pattern, `"MOVE"` and `"TO"` are two terminal segments, `-`, `var_1`, `var_2` are patterns defined before and `@var_1` and `@var_2` are two PDL variables.

Several program understanding tools use the pattern matching engine: search of a pattern, link the execution of a program to a pattern, variable dependency graph (see section 6.3).

The pattern matching searches for a given pattern in a text or in the description of the object of the current schema. The user can ask to search for the next occurrence of the pattern or to select all the occurrences of the pattern in the current document.

It is possible to associate the execution of a procedure to a pattern. Each time a pattern matches, the procedure is executed with the variables of the patterns as input parameters of the procedure. This can be very useful to automate a process: generate a report with the pattern found or creates constraints each time the pattern is found. For example, if we have a SQL database where the views materialize sub-types of tables, the views are defined as follow:

```
create view ....
as select (...)
from <table>
where <column> = <string>;
```

We'd like to create an is-a relation between the table and its views. The SQL extractor will create an entity type for each view and put its definition into the technical description (see section 8.3.2 of chapter 8). So we can search for the following pattern in the technical description of the database schema

```
is-a ::= "from" - @table - "where" - @column ~ "=" ~ @string;
```

and link it to a procedure that will create the is-a relation between the table (variable `table`) and the current entity type.

6.3. Variable dependency graph

In DBRE, it is often useful to know to which other variables (or attributes) an attribute of the database is connected. For example, it can be very useful to refine the attributes decomposition. If an attribute is mapped to a variable that is decomposed in sub-variables, we can conclude that the attribute can be decomposed as the variable. It is useful to have a weak, easy to compute, version of a dataflow diagram, called *variable dependency graph* (VDG). In this graph, each variable of the program is represented by a vertex, while an arc (directed or not) represents a direct relation (assignment, comparison, etc.) between two variables. If there is a path from variable `A` to variable `C` in the graph, then there is, in the program, a sequence of statements such that the value of `A` is in relation with the value of `C`.

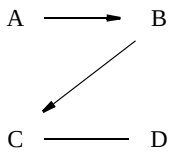
The very meaning of the relation between variables is defined by the analyst depending on the type of relation materialized by the arcs. The interpretation of the variable `A` being in relation with variable `C` can be one of the following: the structure of one variable is a variant of the other one, the

variables share the same values, they denote the same real world object, there is a dependency between the two variables, etc.

To construct this graph, it is only needed to search the program for definite statement patterns. Without worrying to write a complete parser that analyzes the whole program. Figure 62.b illustrates the variable dependency graph of the program fragment shown in figure 62.a.

```
...
MOVE A TO B.
...
MOVE B TO C.
...
IF (C = D)
...
```

a) The code fragment.



b) The VDG.

```
move::="MOVE" - @var_1 - "TO" - @var_2;
compare::=@var_1 - rel_op - @var_2 ;
```

c) The patterns.

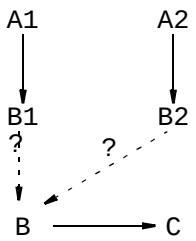
FIGURE 62. The variable dependency graph.

In DB-MAIN, the relation is defined as a pattern in which the two variables in relation are represented by two PDL variables (`var_1` and `var_2`). The patterns used to construct figure 62.b graph are displayed in figure 62.c.

The usage of the variable dependency graph can lead to three kinds of silence and to two kinds of noise.

```
01 A1 PIC X(10).
01 A2 PIC X(10).
01 B.
  02 B1 PIC X(10).
  02 B2 PIC X(10).
01 C  PIC X(20).
MOVE A1 TO B1.
MOVE A2 TO B2.
MOVE B TO C.
```

a) The code fragment.



b) The VDG.

FIGURE 63. Example of silence in variable dependency graph due to variable decomposition unawareness.

The first source of silence lies in the relations that are represented by the arcs. If we use assignment statements only, then all the other instructions that contribute to the dataflow (compute, multiply, string,...) are ignored. This kind of silence can be reduced by increasing the number of statements we are looking for.

The second source of silence is that the graph is not aware of the structure of the variables. Figure 63 gives an example where such silence appears. The decomposition of `B` in `B1` and `B2` is not represented, so that the path between (`A1`, `A2`) and `C` remains undetected.

```
fd A.
 01 REC-A.
   ...
   02 A-I... .
fd B.
 01 REC-B.
   ...
   02 B-J... .
```

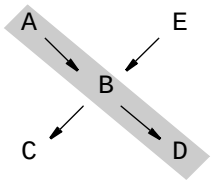
```
0 Main.
1  read A
2  if(A-I = “T”)
3   move “c1” to B-J
   else
4   move “c2” to B-J.
5  write REC-B.
```

FIGURE 64. The dependency between *A-I* and *B-J* is implemented using a test (`if`).

Finally, ignoring control flow can also generate silence. For example in figure 64, the result of the test on `A-I` (`if(A-I="T")`) is necessary to discover the dependency (a computed dependency) between `A-I` and `B-J`. The value of `B-J` is influenced by the value of `A-I`. The corresponding VDG is empty because there is no assignment between variables in this example, but there is a dependency. The last two kinds of silence are very difficult to address with this technique.

```
MOVE A TO B
............
MOVE B TO C
............
MOVE E TO B
............
MOVE B TO D
```

a) The code fragment.



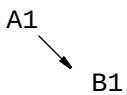
b) The VDG.

FIGURE 65. Example of variable dependency graph noise.

Noise can be generated because the graph only represents dataflow and not the control flow. There exist variables that are connected by a path in the graph though they are not in relation at execution time. As show in figure 65, there is a path between variables `A` and `D` in the graph, but they are not in relation during the execution. Between the assignment for `A` to `B` and the assignment from `B` to `D`, there is an assignment that overwrites the value of `B`.

The second source of noise is that if a variable represents a record field, it does not necessarily contain a value that appears in the database. Let us consider the following tricky program:

```
READ A.
MOVE "cst" TO A1.
MOVE A1 TO B1.
WRITE B.
```



... where `A1` is an attribute of entity type `A` and `B1` of entity type `B`. The graph shows a relation between `A1` and `B1`, so we can conclude erroneously that there is a dependency between the entity types `A` and `B`. The value of `A1` that is assigned to `B2` is not a value stored into entity type `A`, but a constant, so it is erroneous to conclude that there is a dependency between both entity types.

6.4. Program slicing

This section describes the program slicing developed in the DB-MAIN CASE tool. This slicing tool analyzes COBOL program, so it needs to analyze programs with procedures and arbitrary control flow (`go to`).

In the DB-MAIN program slicing tool, a program is represented by a graph (the system dependency graph) and the slicing problem is simply a vertex-reachability problem, and thus slices may be computed in linear time in the number of edges when the graph is already computed. The computation of the graph is more expensive. As said earlier, we are interested in interprocedural slicing with arbitrary control flow, generating a slice of an entire program, where the slice crosses the boundaries of procedural calls and with `go to`'s. For the interprocedural slice, we use uses the system dependency graph to represent the program and the algorithm to compute the slice that was described by S. Horwitz et al. in [Horwitz et al.-1990]. It uses the augmented system dependency graph as proposed by Ball et al. in [Ball et al.-1992] to resolve the orthogonal problem of the slicing of procedure with arbitrary control flow.

The remainder of this section is organized as follows. Section 6.4.1 is a brief state of the art of the different slicing techniques. Section 6.4.2 provides background material, including the definition of control flow graph and program dependence graph. Section 6.4.3 presents the system dependency graph. Section 6.4.4 discusses the slicing algorithm. Section 6.4.5 provides information about the augmented SDG to resolve the problem of procedures with arbitrary control flow. Section 6.4.6 describes how to construct the augmented SDG from the COBOL source text.

6.4.1. Program slicing state of the art

Program slicing is a decomposition technique that extracts from a program the statements relevant to a particular computation. Informally, a slice provides the answer to the question "What program statements do potentially affect the computation of variable *V* at point *p*?"

The *slice* of a program with respect to program point *p* and variable *V* consists of all statements and predicates of the program that might affect the value of *V* at point *p*. This concept, originally discussed by M. Weiser in [Weiser-1984], can be used to debug programs, maintain programs, understand programs behavior. The task of computing program slices is called *program slicing*. Weiser claims that a slice corresponds to the mental abstractions that people make when they are debugging a program.

Various slightly different notions of program slices have been proposed, as well as number of methods to compute slices. Features of programming languages such as procedures, arbitrary control flow, composite data types and pointers each require a specific solution. An important distinction is that between static and dynamic slice. The former notion is computed without making assumptions regarding a program's input, whereas the latter relies on some specific test case.

A complete overview of the difference notion of slicing and of the computation methods can be found in [Binkley et al.-1996] and [Tip-1994].

6.4.1.1. Static slicing

In Weiser's approach, slices are computed by computing consecutive sets of indirectly relevant statements, according to dataflow and control flow dependencies. Only statically available information is used for computing slices; hence, this type of slice is referred to as a *static slice*. In Weiser's terminology, a *slicing criterion* is a pair <*p*,*V*>, where *p* is a program point and *V* is a subset of the program's variables. Computing a slice from a control-flow graph is a two-step process. First requisite dataflow information is computed. The dataflow information is the set of relevant variables at each point *p*. The second step identifies the statements of the slice. These include all points *p* that assign to a variable relevant at *p* and the slice taken with respect-to any predicate statement that directly controls *p*'s execution.

An alternative method for computing static slices was suggested by Ottenstein and Ottenstein [Ottenstein et al.-1994], who restate the problem of static slicing in terms of a reachability problem in a *program dependence graph* (PDG). A PDG is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies. The slicing criterion is identified with a vertex in the PDG, and a slice corresponds to all PDG vertices from which the vertex under consideration can be reached. Various program slicing approaches utilize modified and extended versions of PDGs as their underlying program representation.

The slices mentioned so far are computed by gathering statements and control predicates by way of a backward traversal of the program, starting at the slicing criterion. Therefore, these slices are referred to as *backward* slices. Horwitz et al. were the first who introduced the notion of *forward* slicing in [Horwitz et al.-1990]. A forward slice consists of all statements and control predicates dependent on the slicing criterion.

Interprocedural slicing as a graph reachability problem requires extending the PDG and it also requires modifying the slicing algorithm. [Horwitz et al.-1990] introduced the term *system dependence graph* (SDG) for the dependence graphs that represent multi-procedure graphs. See section 6.4.3 for a complete description of the system dependence graph.

[Ball et al.-1992] and [Choi et al.-1994] present two methods for slicing in the presence of arbitrary control flow (programs containing `go to`'s). Both methods require modifying the control dependence subgraph of the PDG, but not the slicing algorithm. See section 6.4.5 for a complete description of the Ball et al. method.

In the presence of pointers (and procedures), situations may occur where two or more variables refer to the same memory location; this phenomenon is commonly called *aliasing*. Algorithms for determining potential aliases can be found in [Choi et al.-1993] and [Landi et al.-1992]. Slicing in the presence of aliasing requires a generalization of the notion of data dependence to take potential aliases into account.

6.4.1.2. Dynamic slicing

Korel and Laski introduce the notion of *dynamic* slicing [Korel et al.-1988]. In the case of dynamic slicing, only the dependences that occur in a specific execution of the program are taken in account. An alternate view of the difference between static and dynamic slicing is that dynamic slicing assumes a fixed input for a program, whereas static slicing does not make assumptions regarding

the input. The availability of run-time information makes dynamic slices smaller than static slices, but limits their applicability to that particular input.

Agrawal et al. presented the first algorithm for finding dynamic slices using dependence graphs [Agrawal et al.-1991].

Hybrid approaches are a combination of static and dynamic information used to compute slices.

6.4.2. Program dependency graph

Different definitions of program dependence representations were proposed, depending on the intended application, and share the common feature of having an explicit representation of data dependencies (see below). The "program dependence graphs" defined in [Ottenstein et al.-1994] introduced the additional feature of an explicit representation for control dependencies (see below).

The *program dependence graph* (PDG) for program *P*, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edge. The vertices of G_P represent the assignment statements and control predicates that occur in program *P*. In addition, G_P includes three other categories of vertices:

1. There is a distinguished vertex called the *entry vertex*.
2. For each variable *x* for which there is a path in the standard control flow graph for *P* on which *x* is used before being defined, there is a vertex called the initial definition of *x*. This vertex represents an assignment to *x* from the initial state. The vertex is labeled "*x := InitialState(x)*".
3. For each variable *x* named in *P*'s end statement, there is a vertex called the final use of *x*. It represents an access to the final value of *x* computed by *P*, and is labeled "*FinalUse(x)*".

The edges of G_P represent dependencies among program components. An edge represents either control dependence or data dependence. Control dependence edges are labeled either true or false, and the source of a control dependence edge is always the entry vertex or a predicate vertex. A *control dependence edge* from vertex v_1 to vertex v_2 means that, during execution, whenever the predicate represented by v_1, is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will eventually be executed if the program terminates.

A *data dependence edge* from vertex v_1 to vertex v_2 means that the program's computation might be changed if the relative order of the components represented by v_1, and v_2 were reversed.

```
WORKING-STORAGE SECTION.
01 s pic 99.
01 i pic 99.

PROCEDURE DIVISION.
ENTRY.
  MOVE 0 TO s.
  MOVE 1 TO i.
  PERFORM UNTIL i>11
    ADD i TO s
    ADD 1 TO i.
  DISPLAY s.
  DISPLAY i.
  STOP RUN.
```

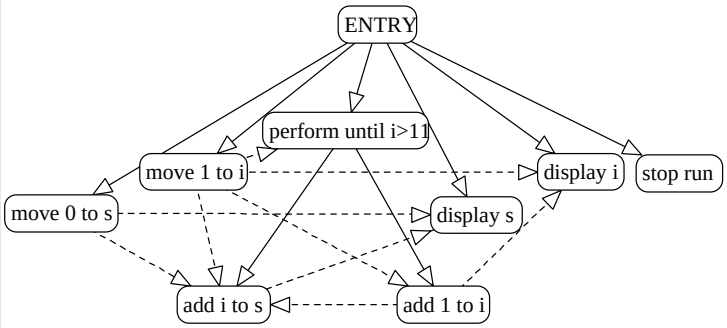


FIGURE 66. A program and its PDG. The solid arrows represent control dependence edges and dashed arrows represent data dependence edges.

Example. Figure 66 shows a program and its corresponding program dependency graphs. The solid arrows represent control dependence edges and dashed arrows represent data dependence edges.

The algorithm to construct the PDG will be presented in section 6.4.6.

6.4.3. The system dependency graph

The *system dependence graph*, an extension of the dependence graphs defined in section 6.4.2, represents programs in a language that includes procedures and procedure calls.

The definition of the system dependence graph (SDG) models a language with the following properties:

1. A complete system consists of a single (main) program and a collection of auxiliary procedures.
2. Procedures end with return statements instead of end statements (as defined in section 6.4.2). A return statement does not include a list of variables.
3. Parameters are passed by value-result.

Further assumptions are made that there are no call sites of the form *P(x, x)* or of the form *P(g)*, where *g* is a global variable. The former restriction sidesteps potential copy-back conflicts. The latter restriction permits global variables to be treated as additional parameters to each procedure; thus, we do not discuss global variables explicitly in this section.

A system dependence graph includes a program dependence graph, which represents the system's main program, procedure dependence graphs, which represent the system's auxiliary procedures and some additional edges. These additional edges are of two sorts: (1) edges that represent direct dependencies between a call site and the called procedure, and (2) edges that represent transitive dependencies due to calls.

Extending the definition of dependence graphs to handle procedure calls requires representing the passing of values between procedures. When procedure *P* calls procedure *Q*, values are transferred

from *P* to *Q* by means of intermediate temporary variables, one for each parameter. A different set of temporary variables is used when *Q* returns to transfer values back to *P*. Before the call, *P* copies the values of the actual parameters into the call temporaries variables; *Q* then initializes local variables from these temporaries variables. Before returning, *Q* copies return values into the return temporaries variables, from which P retrieves them.

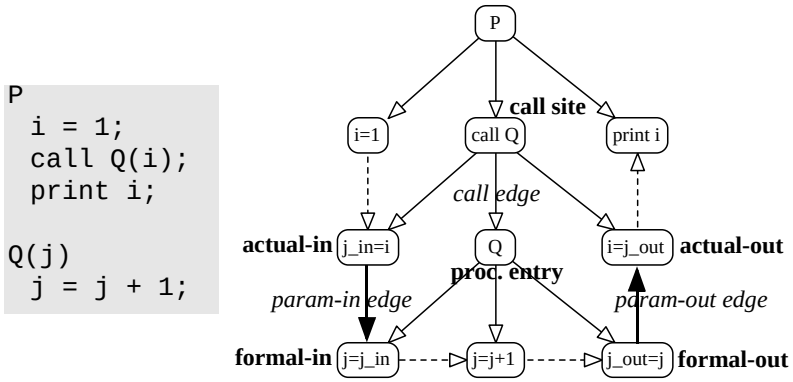


FIGURE 67. SDG with the new kinds of vertices and edges named.

This model of parameter is represented in procedure dependence graphs through the use of five new kinds of vertices (see figure 67). A call site is represented using a *call-site* vertex; information transfer is represented using four kinds of parameter vertices. On the calling side, information transfer is represented by a set of vertices called *actual-in* and *actual-out* vertices. These vertices, which are control dependent on the call-site vertex, represent assignment statements that copy the values of the actual parameters to the call temporaries variables and from the return temporaries variables, respectively. Similarly, information transfer in the called procedure is represented by a set of vertices called *formal-in* and *formal-out* vertices. These vertices, which are control dependent on the procedure's entry vertex, represent assignment statements that copy the values of the formal parameters from the call temporaries variables and to the return temporaries variables, respectively.

Using this model, data dependencies between procedures are limited to dependencies from actual-in vertices to formal-in vertices and from formal-out vertices to actual-out vertices. Connecting procedure dependence graphs to form a system dependence graph is straightforward, involving the addition of three new kinds of edges:

1. A *call edge* is added from each call-site vertex to the corresponding procedure-entry vertex.
2. A *parameter-in* edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure.
3. A *parameter-out* edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

Call edges are a new kind of control dependence edge; parameter-in and parameter-out edges are new kinds of data dependence edges.

Another advantage of this model is that flow dependencies can be computed in the usual way, using data-flow analysis on the procedure's control-flow graph. That graph includes vertices analogous to the actual-in, actual-out, formal-in and formal-out vertices of the procedure dependence graph. A procedure's control-flow graph starts with a sequence of assignments that copy values from call temporaries to formal parameters and ends with a sequence of assignments that copy values from

formal parameters to return temporaries. Each call statement within the procedure is represented in the procedure's control-flow graph by a sequence of assignments that copy values from actual parameters to call temporaries, followed by a sequence of assignments, which copy values from return temporaries to actual parameters.

An important question is which values are transferred from a call site to the called procedure and back again. This point is discussed further in section 6.4.4, which presents a strategy where the results of interprocedural data-flow analysis are used to omit some parameter vertices from procedure dependence graphs. For now, it is assumed that all actual parameters are copied into the call temporaries and retrieved from the return temporaries. Thus, the parameter vertices associated with a call from procedure P to procedure Q are defined as follows (G_P denotes the procedure dependence graph for P):

```
principal.
 move 0 to s.
 move 1 to i.
 perform proc-a until i>11
 stop run.
```

```
proc-a.
  move s to a.
  move i to b.
  perform proc-add.
  move a to s.
  move b to z.
  perform proc-inc.
  move z to i.
```

```
proc-add.
  compute a = a + b.

proc-inc.
  move z to a.
  move 1 to b.
  perform proc-add.
  move a to z.
```

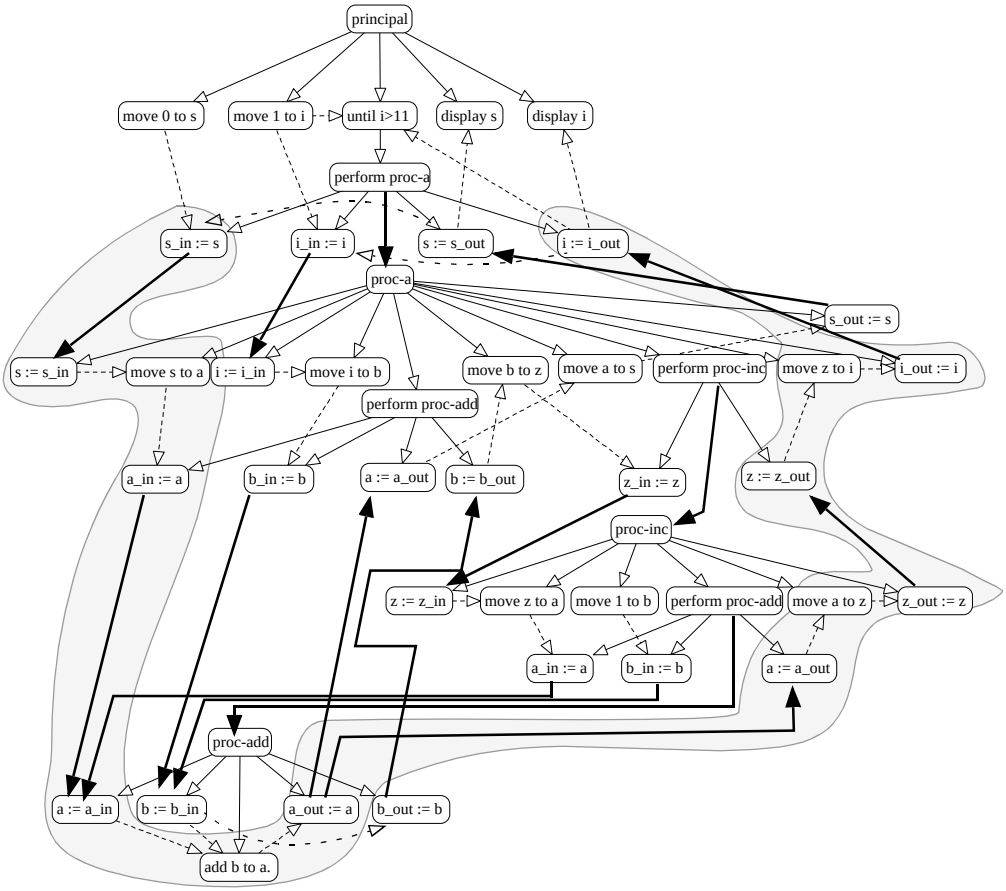


FIGURE 68. A program and its corresponding SDG. The PDG are connected with parameter-in, parameter-out and call edges.

In G_P, subordinate to the call-site vertex that represents the call to Q, there is an actual-in vertex for each actual parameter e of the call to Q. The actual-in vertices are labeled *r_in := e*, where *r* is the formal parameter name.

For each actual parameter *a* that is a variable (rather than an expression), there is an actual-out vertex. These are labeled *a := r_out* for actual parameter *a* and corresponding formal parameter *r.*

The parameter vertices associated with the entry to procedure Q and the return from procedure Q are defined as follows (G_Q denotes the procedure dependence graph for Q):

For each formal parameter *r* of Q, G_Q contains a formal-in vertex and a formal-out vertex. These vertices are labeled *r := r_in* and *r_out := r*, respectively.

Example. Figure 68 shows a COBOL program and its corresponding system dependence graph, connected with parameter-in edges, parameter-out edges and call edges. Edges representing control dependencies are shown as plain lines, edges representing intraprocedural data dependencies are shown using dashed lines; parameter-in edges, parameter-out edges, and call edges are shown using bold lines. In COBOL programs there are only global variables and there is no parameter passed by procedure call (`PERFORM`). Since the SDG graph does not accept global variables, we simulate them through parameters of procedure calls. All the variables that are initialized before the procedure call and that are used by the procedure are represented as formal-in vertex. All the variables that are modified by the procedure and that are used outside the procedure are represented as formal-out vertex.

Using the graph structure defined as far, interprocedural slicing could be defined as a graph-reachability problem, and the slices obtained would be imprecise. This method does not produce as precise slices as possible because it fails to account for the calling context of a called procedure.

Example. This can be illustrated using the graph shown in figure 68. In the graph-reachability vocabulary, the problem is that there is a path from the vertex of procedure *main* labeled "*s_in := s*" to the vertex of *main* labeled "*i: = i_out*", even though the value of *i* after the call to procedure *proc-a* is independent of the value of *s* before the call. The source of this problem is that not all paths in the graph correspond to possible execution paths (e.g., the path, greyed in figure 68, from vertex "*s_in := s*" of *main* to vertex "*i := i_out*" of *main* corresponds to procedure *proc-add* being called by procedure *proc-a*, but returning to procedure *proc-inc*).

To overcome this problem, an additional kind of edge is added to the system dependence graph to represent transitive dependencies due to the effects of procedure calls. The presence of transitive-dependence edges permits interprocedural slices to be computed in two passes, each of which is cast as a reachability problem.

The system dependence graph is constructed by the following steps:

1. For each procedure of the system, construct its procedure dependence graph.
2. For each call site, introduce a call edge from the call-site vertex to the corresponding procedure-entry vertex.
3. For each actual-in vertex *v* at a call site, introduce a parameter-in edge from *v* to the corresponding formal-in vertex in the called procedure.
4. For each actual-out vertex *v* at a call site, introduce a parameter-out edge to *v* from the corresponding formal-out vertex in the called procedure.

5. At all call sites that call procedure *P*, introduce flow dependence edges that represent transitive dependence due to the effects of procedure calls. Such edges are called *def-order edges*.

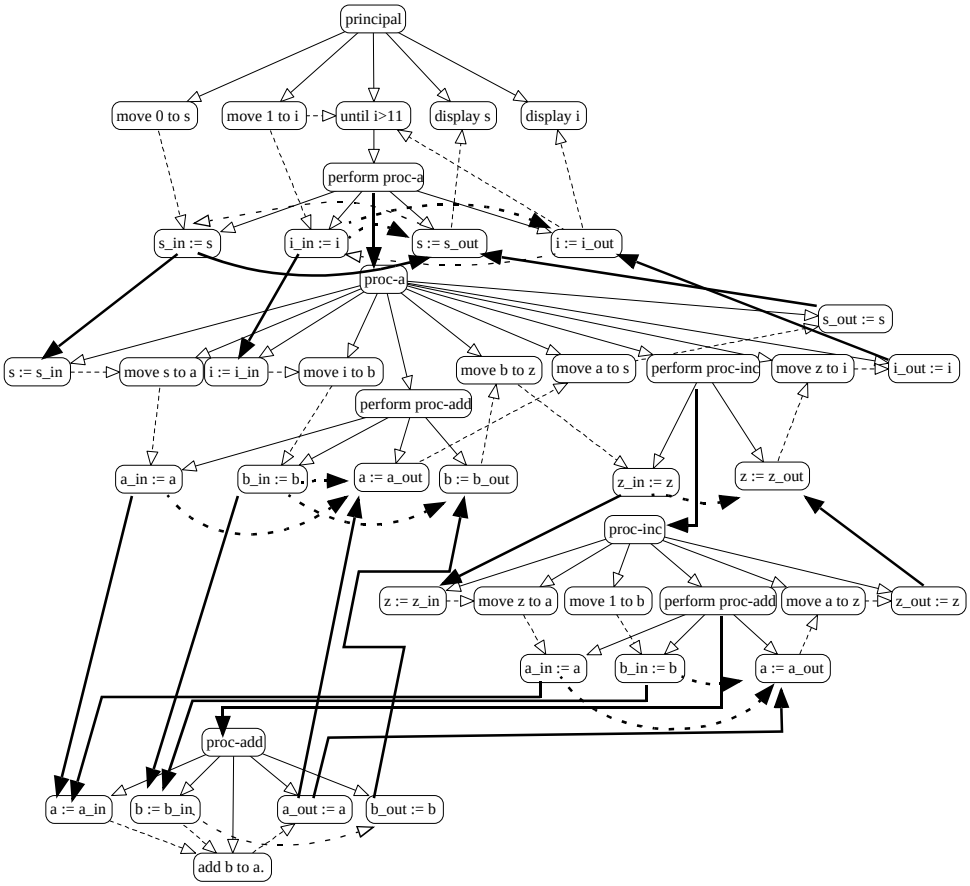


FIGURE 69. The complete SDG of figure 68 program.

Example. Figure 69 shows the complete system dependence graph for figure 68 program. Control dependencies are represented using plain arrows; intraprocedural flow dependencies are represented using dashed arrows; transitive interprocedural data dependencies (corresponding to subordinate characteristic graph edges) are represented using dashed, bold arcs; call edges, parameter-in edges, and parameter-out edges (which connect program and system dependence graphs together) are represented using bold arrows.

The construction of the SDG will be explained in section 6.4.6.

6.4.4. Interprocedural slicing

This section describes how to perform an interprocedural slice using the system dependence graph defined in section 6.4.3.

The difficult aspect of interprocedural slicing is keeping track of the calling context when a slice "descends" into a called procedure.

The key element of this approach is the use of the edges that represent transitive data and control dependencies from actual-in vertices to actual-out vertices due to procedure calls. The presence of such edges permits to sidestep the "calling context" problem; the slicing operation can move "across" a call without having to descend into it.

Suppose the goal is to slice system dependence graph *G* with respect to some vertex *s* in procedure *P*; Phases 1 and 2 can be characterized as follows:

Phase 1. Phase 1 identifies vertices that can reach *s*, and are either in *P* itself or in a procedure that calls *P* (either directly or transitively). It follows flow edges, control edges, call edges and parameter-in edges. Because parameter-out edges are not followed, the traversal in Phase 1 does not "descend" into procedures called by *P.* The effects of such procedures are not ignored, however; the presence of transitive flow dependence edges from actual-in to actual-out vertices (subordinate-characteristic-graph edges) permits the discovery of vertices that can reach *s* only through a procedure call, although the graph traversal does not actually descend into the called procedure.

Phase 2. Phase 2 identifies vertices that can reach *s* from procedures (transitively) called by *P* or from procedures called by procedures that (transitively) call *P.* It follows flow edges, control edges and parameter-out edges. Because call edges and parameter-in edges are not followed, the traversal in Phase 2 does not "ascend" into calling procedures; the transitive flow dependence edges from actual-in to actual-out vertices make such "ascents" unnecessary.

Both Phases 1 and 2 traversed the system dependence graph to find the set of vertices that can reach a given set of vertices along certain kinds of edges. The traversal in Phase 1 follows flow edges, controls edges, call edges, and parameter-in edges, but does not follow parameter-out edges. The traversal in Phase 2 that follows flow edges, control edges, and parameter-out edges, but does not follow, call edges, or parameter in edges.

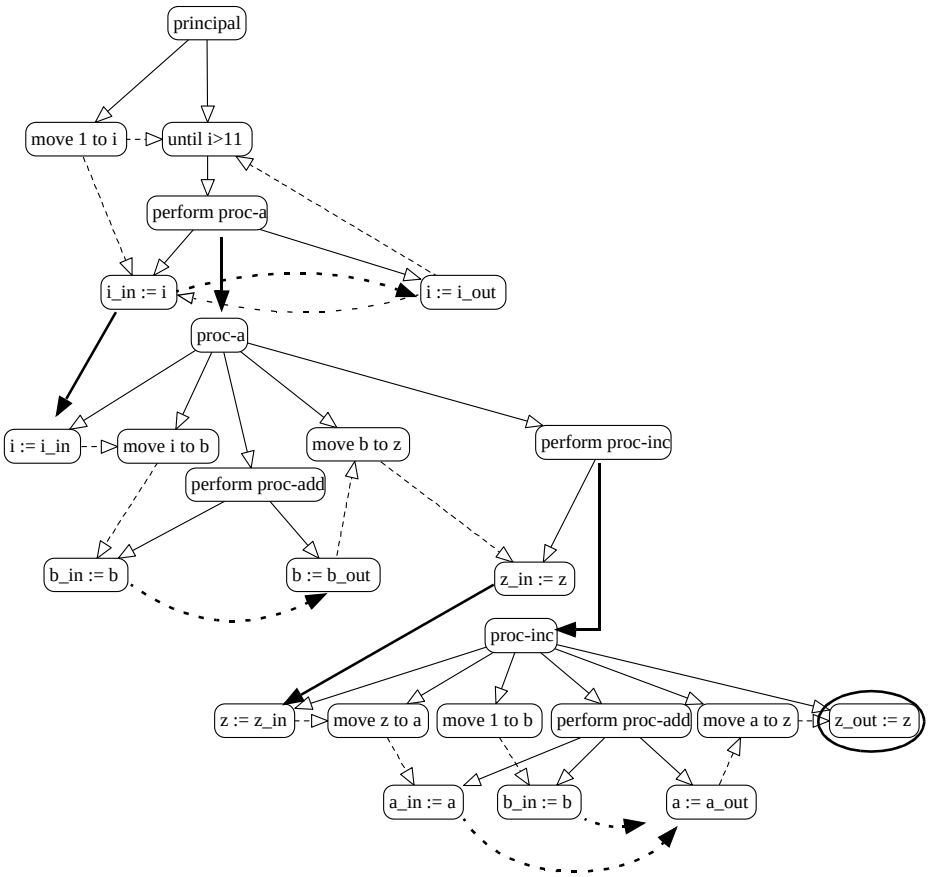


FIGURE 70. The example program's SDG is sliced with respect to the formal-out vertex for parameter z in procedure proc-inc. The vertices marked by Phase 1 of the slicing algorithm as well as the edges traversed during this phase.

Figure 70 and figure 71 illustrate the two phases of the interprocedural slicing algorithm. Figure 70 shows the vertices of the example system dependence graph of figure 69 that are marked during Phase 1 of the interprocedural slicing algorithm when the system is sliced with respect to the formal-out vertex for parameter *z* in procedure proc-inc. Edges "traversed" during Phase 1 are also included in figure 70.

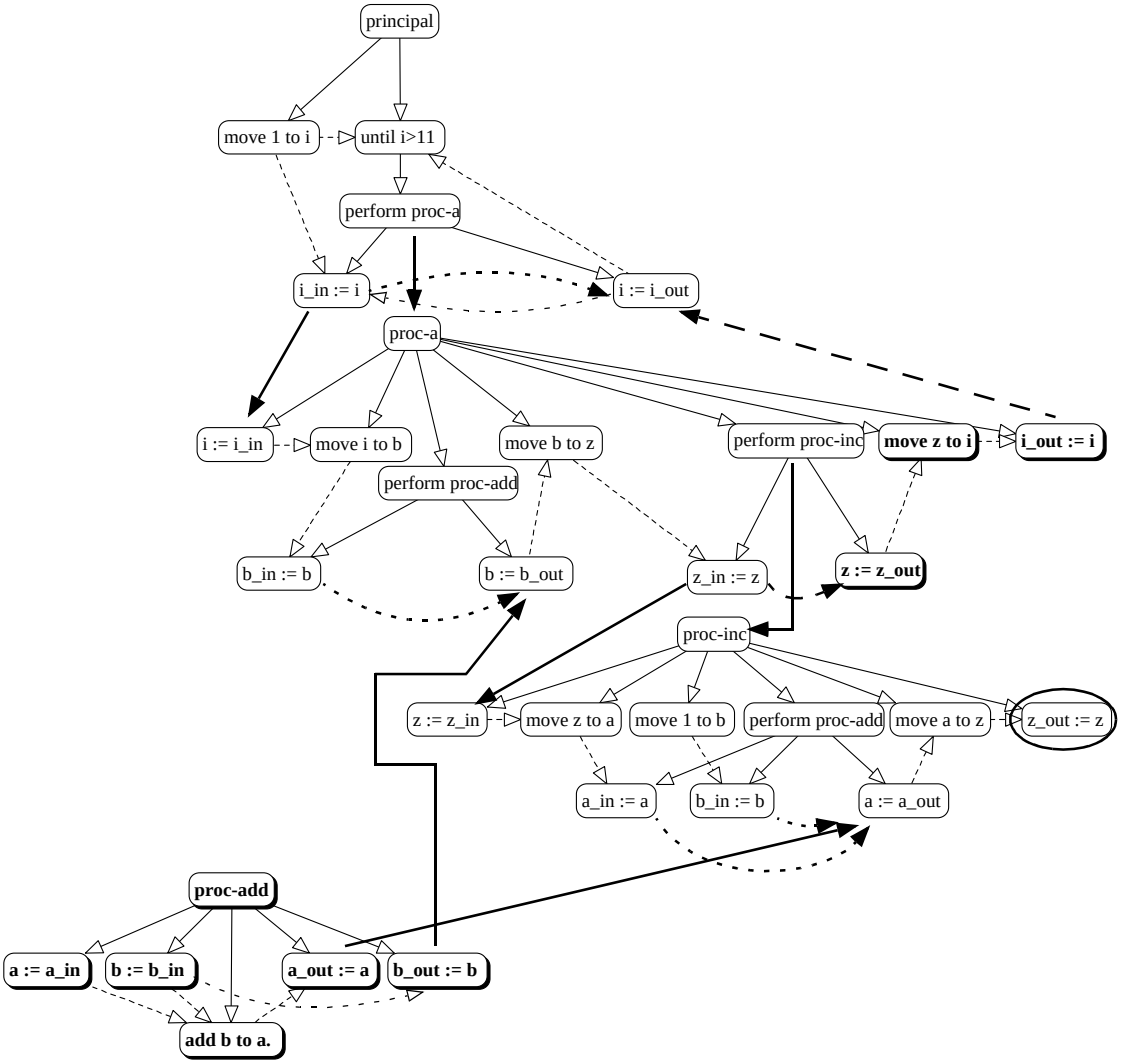


FIGURE 71. The vertices marked by phase 2 of the slicing algorithm as well as the edges traversed during this phase are shown in boldface.

Figure 71 adds (in boldface) the vertices that are marked and the edges that are traversed during Phase 2 of the slice.

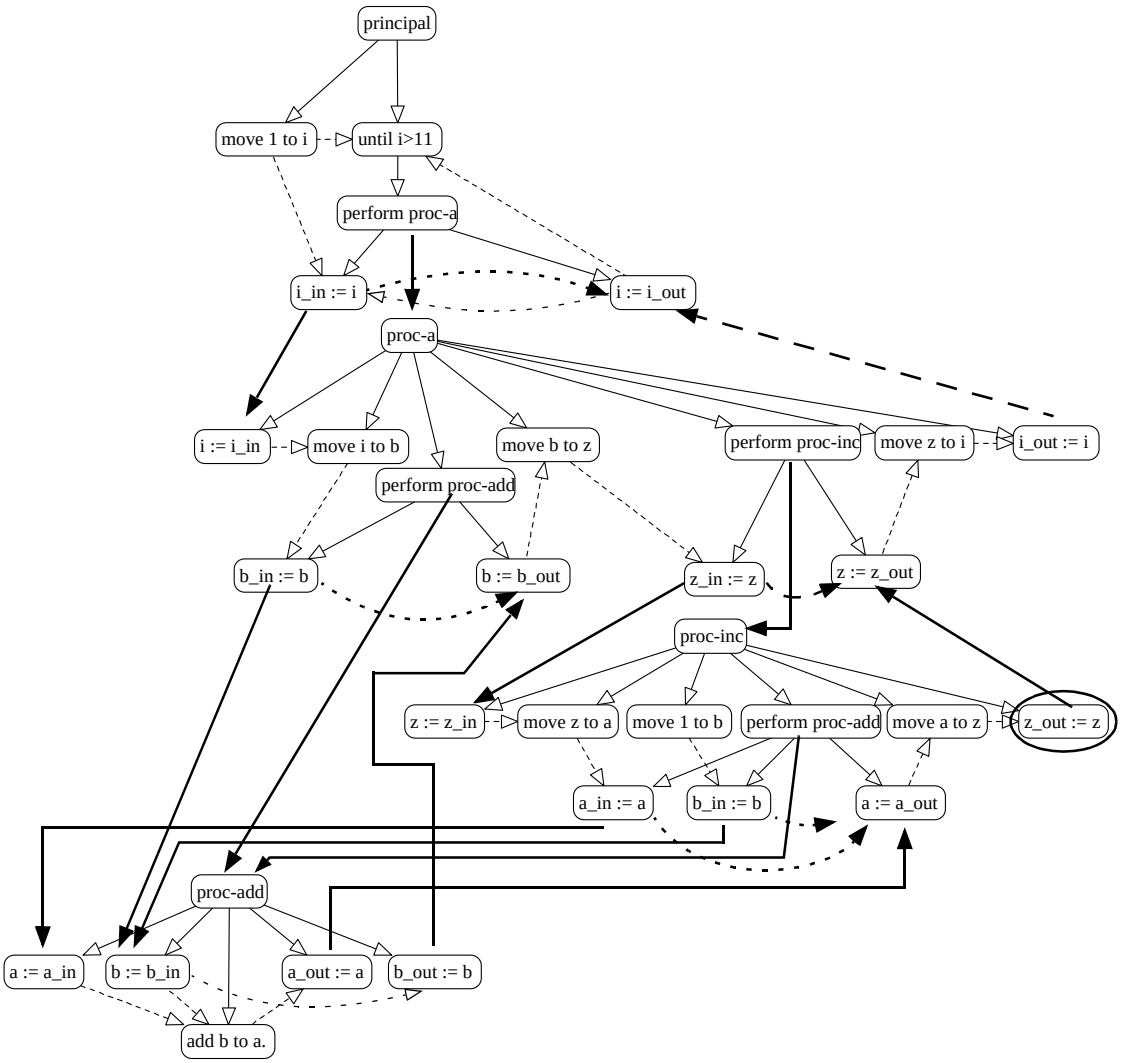


FIGURE 72. The complete slice of the example program's system dependence graph sliced with respect to the formal-out vertex for parameter *z* in procedure *proc-inc*.

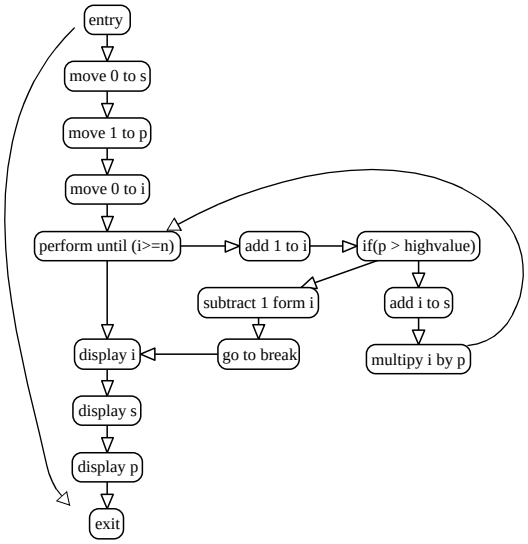
The result of an interprocedural slice consists of the sets of vertices identified by Phase 1 and Phase 2 and the set of edges induced by this vertex set. Figure 72 shows the completed example slice (excluding def-order edges.)

6.4.5. Arbitrary control flow

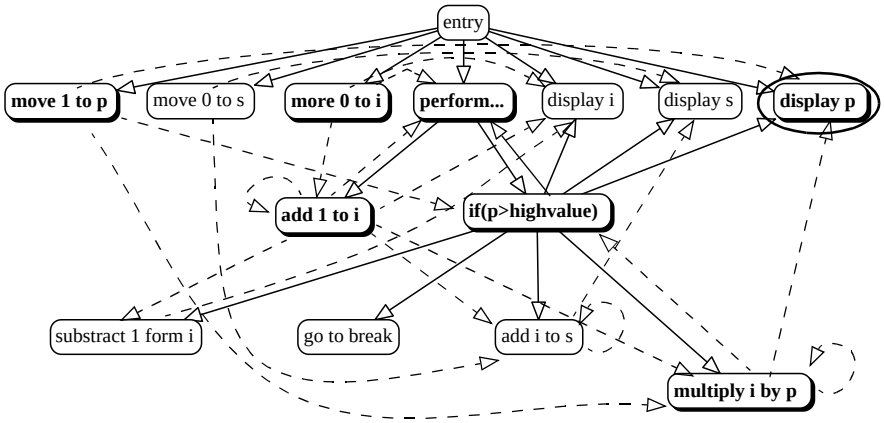
A lot of COBOL programs, especially legacy ones, contain `Go To` statements (arbritary control flow) that are not correctly handled by the algorithm of Horwitz et al. [Horwitz et al.-1990] as shown in the following example. The problem of slicing in presence of arbritary control flow is orthogonal to the multiple procedures problem discussed in the previous sections. The SDG is augmented as suggested by Ball and Horwitz in [Ball et al.-1992].

```
1   move 0 to s.
2   move 1 to p.
3   move 0 to i.
4   perform until (i >= N)
5     add 1 to i
6     if(p > highvalue) then
7       subtract 1 from i
8       go to break
9     end-if
10    add i to s
11    multiply i by p.
12break.
13  display i.
14  display s.
15  display p.
16  stop run.
```

a) The program.



b) The CFG.



c) The PDG.

```
2   move 1 to p.
3   move 0 to i.
4   perform until (i >= N)
5     add 1 to i
6     if(p > highvalue) then
8       go to break
9     end-if
11    multiply i by p.
12break.
15  display p.
```

d) The correct slice.

FIGURE 73. A program with its control flow graph (CFG), its program dependency graph (PDG) and the correct slice with respect to `display p`.

Example. Consider the program shown in figure 73.a. Figure 73.b shows the standard control flow graph (CFG) for this program and figure 73.c shows the program dependence graph that corresponds to this CFG. The bold part of the figure 73.c indicates the vertices that would be identified by our slicing algorithm with respect to the statement *display p*. Figure 73.d shows the correct slice with respect to the statement *display p* with in bold the lines that are not marked by our slicing algo-

rithm. It is clear that the result of the slicing algorithm does not satisfy the semantic goal of program slicing because for some value of *N* and *highvalue*, different final values of *p* will be output by the original program and by the slice.

The problem with the slicing algorithm is that if does not correctly detect when unconditional jumps in the program (such as `go to`) are required in the program projection. Simply including a vertex for the `go to` in the CFG does not solve the problem. The slicing algorithm will still omit the `go to` because there is no path from the `go to` vertex to the vertex that represent the statement `display p`. In fact, the `go to` vertex has no outgoing edges, so it will not be included in any slice other than the slice with respect to the `go to` itself.

The new slicing algorithm itself is similar to the previous one in that it uses the program dependency graph to identify the program components in the slice.

The important difference between the two algorithms is that the new one uses an *augmented control flow graph* (*ACFG*) from which the PDG is build. In particular, jump statements are explicitly represented as a pseudo-predicate vertex. The jump vertex's true-successor is the target of the jump and its false-successor is the vertex that represents the jump statement's continuation (that is, the vertex that would be the jump vertex's successor if it was a "no-op" rather than a jump).

Representing a jump statement this way causes it to be the source of control dependence edges in the PDG. This in turn allows the jump vertex to be included in the set identified by the slicing algorithm.

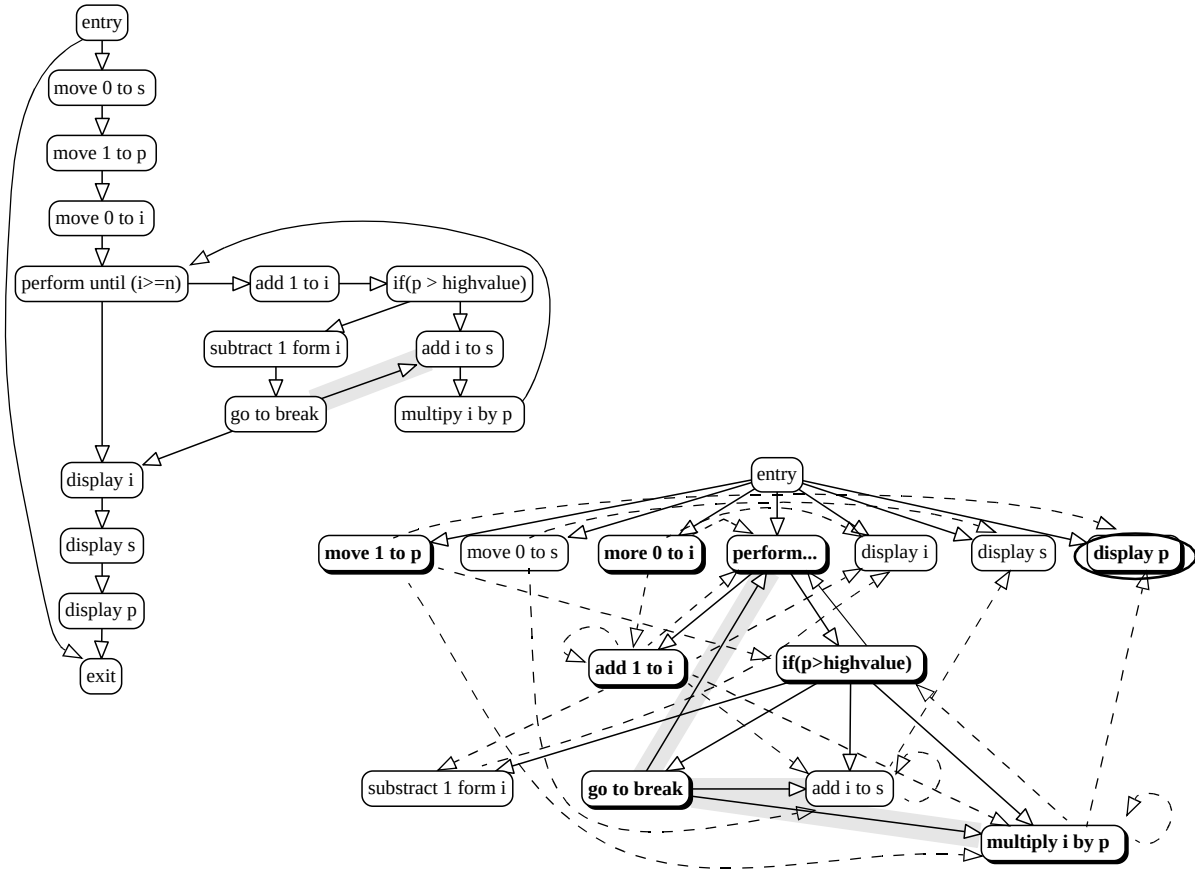


FIGURE 74. The augmented CFG and its corresponding PDG.

Example. The augmented CFG and the correct PDG of figure 73.a program are displayed in figure 74. The bold part of the figure indicates the vertices that would be identified by our new slicing algorithm with respect to `display p`.

6.4.6. SDG construction

A slicing algorithm based on Horwitz et al. [Horwitz et al.-1990] and Ball and Horwitz [Ball et al.-1992] has been presented. During this presentation, it has been sketched that the slicing problem is quite easy (graph traversal) when the SDG is used, but the SDG computation is not a trivial task. This section will present how to compute this SDG from a COBOL program source code.

The different steps of the SDG construction are the following:

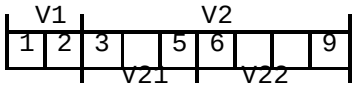
1. COBOL program parsing to obtain the Abstract Syntax Tree (AST).
2. Augmented control flow graph (ACFG) construction from the AST.
3. Computation of the post dom graph from the ACFG.
4. Construction of the PDG using the ACFG and the post dom graph.
5. Construction of the SDG using the PDG.

```
01 cus-address pic X(100).
01 wk-address.
  02 wk-street pic X(60).
  02 wk-city pic X(30).
  02 wk-zip pic X(10).
```

```
move cus-address to wk-address.
display wk-street.
```

FIGURE 75. Example of the need to manipulate part of a variable (`cus-address`).

```
01 V
  02 V1 PIC X(2).
  01 V2.
    03 V21 PIC X(3).
    03 V22 PIC X(4).
```



variable	ancestor	start	stop
V	V	1	9
V1	V	1	2
V2	V	3	9
V21	V	3	5
V22	V	6	9

FIGURE 76. Variable declaration and its internal representation.

6.4.6.1. General consideration

This section will present some general considerations about how we represent variables.

During the SDG construction and its querying, variables comparison is needed (check if a variable is included in another one) to compute variable unions and intersections. Variable names are not necessary, except to display the results. Sometimes, only a part of a variable needs to be manipulated. For example, in figure 75, `display wk_street` depends on the 60 first characters of `cus-address`.

To ease those computations, the variables are represented as their physical position relatively to their ancestor (level 01 variable). Figure 76 gives an example of a variable declaration and its internal representation.

6.4.6.2. The abstract syntax tree

The *abstract syntax tree* (AST) represents the program as a tree where each vertex represents an instruction. The (oriented) arcs represent the syntactic or nested successor of the instruction, vertex can have one or two successors. If a vertex has two successors, the arcs are labeled true and false.

It is not necessary to be able to reconstruct the original program from its control flow graph or its system dependency graph. So the instructions representation can be simplified. There are only 10 types of instructions:

- *Read*: instruction that reads a file.
- *Write*: instruction that writes or modifies a file (as `WRITE` or `REWRITE`).
- *If*: a test.
- *While*: a loop.
- *Call*: a procedure call (COBOL `PERFORM`).
- *Paragraph*: a COBOL paragraph name.
- *Section*: a COBOL section name.

- *End*: an instruction that terminate the program (`STOP RUN`).
- *go to*: a `GO TO` instruction.
- *Normal*: all the other instructions.

For each instruction, there are two lists of variables: the ones that are used (referenced) by the instruction (*Ref*) and the list of the variables that are or may be affected (defined) by the instruction (*Def*).

In the abstract syntax tree and in the CFG each instruction has a pointer to its next instruction (the next instruction of the *go to* is the instruction that follows syntactically). The *while*, *call* and *go to* instructions have a pointer to the next instruction executed (to the loop body for the while). The *read*, *write* and *if* instructions have two next instructions (the true and false successor).

6.4.6.3. The augmented control flow graph

A *control flow graph* (CFG) is a directed graph that satisfies the following conditions. The CFG has three types of vertices: Fall-through vertices (either assignment statements or output statements), which have one successor, predicate vertices, which have one true-successor and one false-successor, and an EXIT vertex, which has no successors. The root of the CFG is the ENTRY vertex, which is a predicate that has the EXIT vertex as its false-successor. Every vertex is reachable from the ENTRY vertex, and the EXIT vertex is reachable from every vertex. Edges in the CFG are labeled; the outgoing edges of a predicate vertex are labeled true or false (as appropriate) and the outgoing edge of a fall-through vertex is labeled null.

The augmented control flow graph (ACFG) [Ball et al.-1992] is a CFG in which a Go To is represented as a pseudo-predicate vertex (that always evaluates to true). The Go To vertex's true-successor is the target of the jump, and its false-successor is the vertex that represents the jump statement's continuation (that is, the vertex that would be the jump vertex's successor if it was a "no-op" rather that a jump).

To compute the ACFG of each procedure, we parse the program to obtain an abstract syntax tree (AST) of the program using the classical compiler techniques [Aho et al.-1989]. This AST can be easily transformed into the ACFG.

In a COBOL program a procedure can be a section or a paragraph, but a paragraph or a section can also represent a label that is the target of a *go to* statement. So the same instruction can belong to several execution paths. We have decided, to facilitate the computation, to duplicate such code, so the same instruction is part of the ACFG of the section and of the ACFG of the paragraph.

```
SECTION S1.
P1.
  PERFORM S2.
...
SECTION S2
P2.
  ...
P3.
...
  PERFORM P2.
```

a) The original program.

```
PROCEDURE S1.
  PERFORM S2.
...
PROCEDURE S2.
...
  PERFORM P2.
PROCEDURE P2.
...
```

b) Program in which P2 code has been duplicated.

FIGURE 77. Program with a paragraph that is part of a section and called as an independent procedure.

Example. Consider figure 77.a program in which the paragraph `P2` is executed as a part of section `S2` (`PERFORM P2`). To facilitate the computation of the SDG the representation of `P2` is duplicated in the AST (figure 77.b).

```
proc =
  Procedure(seq){
    proc.entry=Pred("ENTRY",seq.entry, seq.cont
    proc.cont=seq.cont=FallThrough("EXIT", null
  };

seq =
  NullSeq(){
    seq.entry = seq.cont
  }
| Sequence(stmt seq1){
    seq.entry = stmt.entry
    stmt.cont = seq1.entry
    seq1.cont = seq.cont
  };

stmt =
  While(exp seq) {
    stmt.entry = Pred("expr", seq.entry, stmt.cont)
    seq.cont = stmt.entry
  }
| Normal(expr){
    stmt.entry = FallThrough("expr", stmt.cont)
  }
| IfThen[1](expr seq){
    stmt.entry = Pred("expr", seq.entry, stmt.cont)
    seq.cont = stmt.cont
  }
| IfThenElse[2](expr seq1 seq2){
    stmt.entry = Pred("expr",seq1.entry,seq2.entry)
    seq1.cont = seq2.cont = stmt.cont
  }
| Paragraph(ID) {
    stmt.entry = stmt.cont
    insert(ID, stmt.entry)
  }
| GoTo(ID){
    stmt.entry =
      Pred("GoTo ID", lookup(ID), stmt.cont)
  }
| Call(ID){
  stmt.entry = FallThrough("call ID", stmt.cont)
  stmt.call = ID
  }
```

1. IfThen represents a *If*, a *Read* or a *Write* with only a true-successor without false-successor.
2. IfThenElse represents a *If*, a *Read* or a *Write* with a true-successor and a false-successor.

FIGURE 78. Abstract syntax for the AST with attributes that define the ACFG.

Figure 78 presents an attribute grammar for our AST [Ball et al.-1992], in which the attributes are used to define the translation from an AST to its ACFG. Each production in the grammar is of the form "$x_0 = op(x_1 \ldots x_k)$", where *op* is an operator name and each x_i is a non-terminal. Every non-terminal has two synthesized attributes *entry* and *call* and an inherited attribute *cont*. *Entry* and *cont*

represent vertices in the ACFG and *call* is only valued for the **Call** vertex; it is used to store the name of the called procedure. The constructor $pred(t, v, w)$ creates a predicate vertex with text *t* and true-successor *v* and false-successor *w*, while the constructor $FallThrough(t, v)$ creates a fall-through vertex with text *t* and successor *v*. A global symbol table (with operators *insert* and *lookup*) is used to manage the control flow between **Go To** and **Paragraph**.

$$D(n_e) = \{n_e\}$$
$$\forall (n \in N - \{n_e\}) \text{ do } D(n) = N$$

until one of the $D(n)$ change

$$\forall (n \in N - \{n_e\}) \text{ do}$$
$$D(n) = \{n\} \cup \bigcap_{p \in succ(n)} D(p)$$

$$\forall (n \in N) \text{ do } D(n) = D(n) - \{n\}$$

FIGURE 79. Algorithm to compute the post-dominates.

6.4.6.4. The post-dominators computation

Let *v* and *w* be vertices in an ACFG. Vertex *w post-dominates v* iff $v \neq w$ and *w* is on every path from *v* to the *Exit* vertex.

The algorithm to compute the post-dominance is based on the algorithm to compute dominance [Aho et al.-1989]. If *N* is the set of the vertices of the ACFG and n_e is the *Exit* vertex of the graph then at the end of the algorithm of figure 79, *d* belongs to $D(n)$ iff *d post-dominates n*.

n	*D*(*n*)
16:Exit	
15:display p	16
14:display s	15, 16
13:display i	14, 15, 16
11:multiply i by p	4, 13, 14, 15, 16
10:add i to s	4, 11, 13, 14, 15, 16
8: go to break	13, 14, 15, 16
7: subtract 1 from i	8, 13, 14, 15, 16
6: if(p > highvalue)	13, 14, 15, 16
5: add 1 to i	6, 13, 14, 15, 16
4: perform until (i>=N)	13, 14, 15, 16
3: move 0 to i	4, 13, 14, 15, 16
2: move 1 to p.	3, 4, 13, 14, 15, 16
1: move 0 to s	2, 3, 4, 13, 14, 15, 16
0: Entry	16

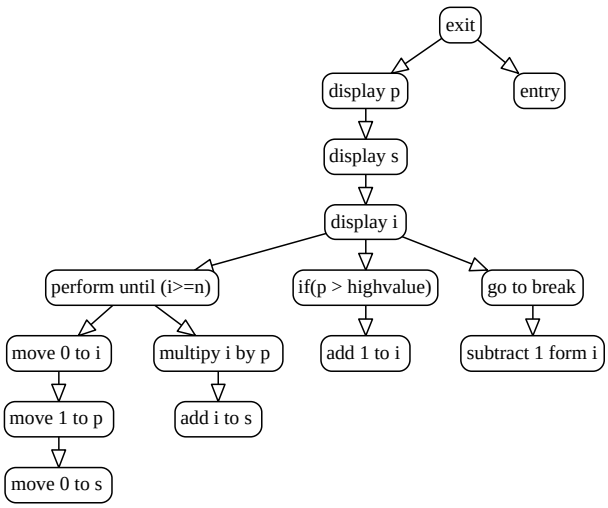


FIGURE 80. The post-dominator computed by the algorithm and the post-dominator tree.

Example. Figure 80 is the results of the algorithm applied to figure 74 ACFG.

6.4.6.5. The PDG construction

The PDG computation is divided in two parts. The control dependencies and the data dependencies computation.

Control dependencies. Given the post-dominator tree, we can determine control dependencies by examining certain ACFG edges and annotating vertices on corresponding post-dominator tree paths [Ferrante et al.-1987]. Let *S* consist of all edges (*A*,*B*) in the ACFG such that *B* is not an ancestor of *A* in the post-dominator tree (i.e., *B* does not post-dominate *A*). Let *L* denote the least common ancestor of *A* and *B* in the post-dominator tree. By construction, we cannot have *L* equal *B*.

Either *L* is *A* or *L* is the parent of *A* in the post-dominator tree.

- *L* = parent of *A*. All vertices in the post-dominator tree on the path from *L* to *B*, including *B* but not *L*, are control dependent on *A*.
- *L* = *A*. All vertices in the post-dominator tree on the path from *A* to *B*, including *A* and *B*, are control dependent on *A*.

After all edges in *S* have been examined, all control dependencies have been determined.

C = the ACFG's arcs; N = the ACGF's vertices
T = post-dominator tree's arcs
$S = \{(A, B) \mid (A,B) \in C \text{ and } (A,B) \notin T\}$
$\forall n \in N$ do $D(n) = \varnothing$

$\forall (A,B) \in S$
$L = predecessor(A)$ in T
$D(A) = \{n \mid n \text{ on the path from } L \text{ to } B \text{ in T and } B \neq L\}$

FIGURE 81. The control dependencies computation.

The algorithm of figure 81 computes $D(n)$ [Aho et al.-1989] as the set of the vertices control dependent on *n* ([Aho et al.-1989]).

S	vertices marked	control dependent on
(0,1)	1,2,3,4,13,14,15	0: Entry
(4,5)	5,6	4: perform until (i >= N)
(6,7)	7,8	6: if(p > highvalue)
(6,10)	4,10,11	6: if(p > highvalue)
(8,10)	4,10,11	8: go to break

FIGURE 82. The vertices marked control dependent and the control dependence edges.

Example. Figure 82 represents the vertices marked by our algorithm for each couple of *S*.

Data dependencies. To compute the data dependencies edges in the PDG, we need to know for each vertex, *n*, the vertices that define the variables referenced in vertex *n* (*n.def*). The dataflow information can be expressed as the following equation [Aho et al.-1989]:

$$\text{Out}(n) = \text{Prod}(n) \cup (\text{In}(n) - \text{Supp}(n))$$

and it can be interpreted as "information at the end of a vertex (Out(*n*)) are produced inside the

vertex (Prod(*n*)) or are present before the vertex (In(*n*)) and are not suppressed (Supp(*n*)). In our implementation Prod(*n*) = Supp(*n*) = *n.def*.

The vertices that define the variables referenced in a vertex *n* (*n.ref*) are $\mathrm{In}(n) \cap n.ref$.

In(*n*) can be computed, if the ACFG (without the false-successor of *Go to*) is known, using the following equation:

$$\mathrm{In}(n) = \bigcup_{p \in Pred(n)} \mathrm{Out}(p)$$

N = ACFG without the Go To false-successors
$\forall n \in$ N do
    Out(*n*) = *n.def*

while some Out(*n*) change
    $\forall n \in$ N do
        $\mathrm{In}(n) = \bigcup_{p \in Pred(n)} \mathrm{Out}(p)$
        Out(*n*) = Out(*n*) $\cup$ (In(*n*) – *n*.def)

FIGURE 83. In and Out computation algorithm.

The algorithm in figure 83 computes In and Out for each vertex of the ACFG.

n	*n.def*	*n.ref*	In(*n*)	Out(*n*)
1: move 0 to s	s			s(1)
2: move 1 to p.	p		s(1)	p(2)s(1)
3: move 0 to i	i		p(2)s(1)	i(3)p(2)s(1)
4: perform until (i>=N)		i	i(3,5)p(2,11)s(1,10)	i(3,5)p(2,11)s(1,10)
5: add 1 to i	i	i	i(3,5)p(2,11)s(1,10)	i(5)p(2,11)s(1,10)
6: if(p > highvalue)		p	i(5)p(2,11)s(1,10)	i(5)p(2,11)s(1,10)
7: subtract 1 from i	i	i	i(5)p(2,11)s(1,10)	i(7)p(2,11)s(1,10)
8: go to break			i(7)p(2,11)s(1,10)	i(7)p(2,11)s(1,10)
10:add i to s	s	i,s	i(5)p(2,11)s(1,10)	s(10)i(5)p(2,11)
11:multiply i by p	p	i,p	s(10)i(5)p(2,11)	p(11)s(10)i(5)
13:display i		i	i(3,5,7)p(2,11)s(1,10)	i(3,5,7)p(2,11)s(1,10)
14:display s		s	i(3,5,7)p(2,11)s(1,10)	i(3,5,7)p(2,11)s(1,10)
15:display p		p	i(3,5,7)p(2,11)s(1,10)	i(3,5,7)p(2,11)s(1,10)

FIGURE 84. In and Out computed by figure 83 algorithm (in 6 iterations).

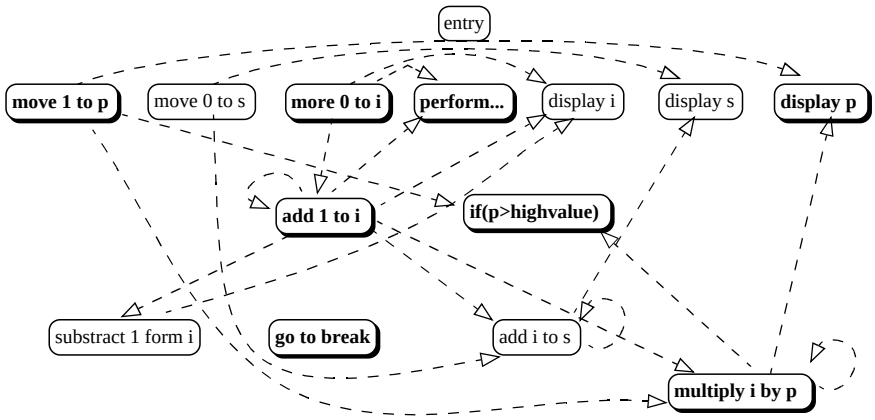


FIGURE 85. PDG's data dependencies edges of figure 66 program.

Example. The result of the algorithm applied to figure 74 ACFG (without the false-successor of Go to) is given in the figure 84 and the PDG's data dependence arcs are shown in figure 85.

6.4.6.6. The SDG construction

When the PDG of each procedure has been constructed, the construction of the SDG is quite easy. The SDG can be constructed as follows:

- For each call site (call vertex in the PDG), a call edge is added from the call site vertex to the corresponding procedure entry vertex. The actual-in and actual-out vertices are connected to the call site vertex.
- For each actual-in vertex *v* at the call site, introduce a parameter-in edge from *v* to the corresponding formal-in vertex in the called procedure.
- For each actual-in vertex *v* at the call site, introduce a parameter-out edge to *v* from the corresponding formal-out vertex in the called procedure.
- At all call sites that call procedure *P,* introduce flow dependence edges that represent transitive dependences due to effects of procedure calls. The flow dependence edges are quite easy to compute for COBOL program because the is no recursivity in the procedure calls. So the procedures can be sorted to compute the flow dependence edges.

6.4.6.7. The complexity of the slicing algorithm

The complexity of the slicing algorithm must be divided in two parts. The first part is the complexity of the SDG construction and the second one is the complexity of the slicing algorithm itself once the SDG is computed. They are dissociated because the SDG is computed only once and stored to compute all the slices needed.

A. Complexity of the SDG construction

The complexity of the SDG construction is the sum of the complexity of each step of its construction as presented in section 6.4.6.

The cost of constructing the SDG can be expressed in terms of the parameters given hereafter:

- #LOC The largest number of LOC in a single procedure.
- #Params The largest number of formal parameters in any procedure.
- #VAR The largest number of variable in any procedure.
- #E The largest number of edges in a single PDG.
- #P The number of procedure in the system.

The AST construction complexity

The complexity of the construction of the AST of a single procedure is linear in the number of LOC.

$$O(AST)=O(\#LOC)$$

The ACFG construction complexity

The complexity of the construction of the ACFG of a single procedure is linear in the number of vertices in the AST, thanks to the use of the attributed grammar. The number of vertices in the AST is of the same order as the number of LOC.

$$O(ACFG)=O(\#LOC)$$

The post-dominator tree construction complexity

The algorithm to compute the post-dominator tree of a single procedure is given in figure 79. This algorithm can be divided in three steps (the initialization, the iteration and the termination). The complexities of the initialization and of the termination are linear in the number of vertices of the ACFG. The number of vertices of the ACFG is of the same as the number of vertices of the AST, which is of the same order as the number of LOC. So the complexity is linear in the number of LOC.

The complexity of the inner loop of the iteration is also linear in the number of vertices of the ACFG. The complexity of the iteration itself (*until one of the D(n) change*) can be evaluated as follows. If a *D(n)* changes then its number of elements decreases, because it is the intersection of elements. During the initialization each *D(n)* has #LOC elements. At each iteration at least one *D(n)* is modified, i.e., its number of elements decreases at least by one. In the worse case the loop exist when all the *D(n)* are empty. So the complexity of the outer-loop is $O(\#LOC^2)$.

The total complexity of the post-dominator tree construction of a single procedure is

$$O(\text{post-dominator}) = O(\text{initialization}) + O(\text{iteration}) + O(\text{termination})$$

$$= O(\#LOC) + O(\#LOC^3) + O(\#LOC)$$

$$=O(\#LOC + \#LOC^3)$$

The control dependencies construction complexity

The algorithm to compute the control dependencies of a single procedure is given in figure 81. The size of *S* is at most the number of edges of ACFG. Each vertices of ACFG has at most two succes-

sors (a true and a false branch). So the size of *S* is at most two times the number of vertices of ACFG (= $2 \cdot \#LOC$).

The complexity of the construction of *D(A)* is linear in the size of the post-dominator tree (= $O(\#LOC)$). The complexity of the construction of the control dependencies for a single procedure is

$$O(\text{control dependencies}) = O(2 \cdot \#LOC \cdot \#LOC)$$

$$= O(\#LOC^2)$$

The data dependencies construction complexity

The algorithm to compute the data dependencies of a single procedure is given in figure 83. The complexity of the initialization is linear in the size of the ACFG (=O(#LOC)).

The maximum number of elements of *in(n)* for any *n* is equal to $\#VAR \cdot \#LOC$, because at most all the variables can be defined in each vertex. If an *in(n)* changes, its number of elements increases by one, because it is an union. So the complexity of the iteration is

$$O(\#VAR \cdot \#LOC \cdot \#LOC)$$

$$= O(\#VAR \cdot \#LOC^2)$$

because each *in(n)* has at most ($\#VAR \cdot \#LOC$) values and at each iteration at least one of the *in(n)* is incremented by one and there is #LOC *n*.

So

$$O(\text{Data dependencies}) = O(\#LOC + \#LOC^2 \cdot \#VAR)$$

The PDG construction complexity

The complexity of the PDG construction is the sum of the complexity of the above graph constructions, i.e., AST, ACFG, post-dominator tree, control dependencies, data dependencies.

$$O(PDG)=O(AST) + O(ACFG) + O(\text{post-dominator}) + O(\text{control dependencies}) + O(\text{data dependencies})$$

$$= (O(\#LOC) + O(\#LOC) + O(\#LOC + O(\#LOC^3)) + O(\#LOC^2) + O(\#LOC + \#LOC^2 \cdot \#VAR))$$

$$= O(\#LOC + \#LOC^2 + \#LOC^3 + \#LOC^2 \cdot \#VAR)$$

The SDG construction complexity

The complexity of the SDG construction is the complexity of each PDG construction, plus the complexity of the creation of the transitive dependence edges.

To determine the dependencies among the parameters of a procedure, the slice with respect to each formal-out is computed. These dependencies are copied to the actual-out to produce the transitive

edges. The complexity to compute the slice with respect to a formal-out is linear in the size of the PDG (O(#E + #LOC)), because the slice is computed by the traversal of the PDG.

The complexity to compute the dependencies among of the parameters of a procedure is

$$O(\#Params \cdot (\#LOC + \#E)$$

The complexity to compute the SDG is

$$O(SDG)=\#P \cdot O(PDG) + O(\#P \cdot \#Params \cdot (\#LOC + \#E))$$

$$=O(\#P \cdot (\#LOC + \#LOC^2 + \#LOC^3 + \#LOC^2 \cdot \#VAR + \#Params \cdot (\#LOC + \#E)))$$

B. Complexity of the slice computation

An interprocedural slice is performed by two traversals of the SDG, starting from some initial set of vertices. The cost of each traversal is linear in the size of the SDG

$$O(slicing)=O(\#P \cdot (\#LOC + \#E))$$

6.5. The program slicing for embedded code

Many legacy COBOL applications do not only use standard files, but also store and manipulate data from DMS. To understand the programs and to recover the implicit constraints of these databases, it is necessary to make the link between the program variables and the database entity types and attributes.

For IDMS/CODASYL DMS, the COBOL language provides built-in instructions (`GET`, `STORE`, `FIND`,...) to access the database and thus those instructions are already analyzed by the program slicing. The only things to add is an option in the program slicing to load the definition of the physical schema referenced by the `SUB-SCHEMA SECTION` of the `DATA DIVISION`. This physical schema is obtained through the DDL analysis.

For other DMS, such as SQL or IMS, COBOL does not have built-in data access instructions. To access those DMS, the programmer writes *embedded instructions* in the program. An embedded instruction belongs to another language, in this case the DMS-DML, that is used in the main program language, called the *host language*. The DMS manufacturer provides a pre-processor (pre-compiler) that translates those embedded DMS-DML instructions into COBOL calls to external functions (programs) that implement the access to the DMS. The difficulties with the embedded instructions is that they are not standardized and thus vary from one DMS to the other and they do not conform to the host language syntax. For example, in an SQL embedded instructions block, each instruction is ended by a ";" while in a COBOL program there is no explicit instruction separator. In SQL, identifiers can contain "_" but no "-" while the reverse holds in COBOL.

Another difficulty, when analyzing programs with embedded code, is that the physical schema is not explicitly declared in the program itself. But the physical schema is necessary to understand the instructions that access the database. The physical schema is obtained through the analysis of the DDL. To compute program slices for a program that contains embedded code, the SDG has to be

produced. This SDG must represent control and dataflow of the COBOL and of the embedded language. To produce such a SDG, a parser that understands COBOL and the embedded language has to be written. To get around these difficulties, we propose to use a preprocessor that translates the embedded instructions into their equivalent COBOL instructions. These instructions are equivalent in that the SDG computed with those instructions or with the embedded instructions is the same. To stress the fact that those instructions do not really implement the embedded code but give an equivalent SDG, two new instructions are added to the COBOL grammar [Sellink et al.-2000].

- `DIRECT-MAP var-1 TO var-2`

  `DIRECT-MAP` has the same behavior as the COBOL `MOVE`, where `var-1` and `var-2` must be of compatible type, the value of `var-1` is assigned to `var-2`.
- `INDIRECT-MAP list-var-1 TO list-var-2`

  All the variables of `list-var-1` are referenced and all the variables of `list-var-2` are defined by the instruction `INDIRECT-MAP`. But the values of the variables of `list-var-1` are not assigned to the variables of `list-var-2`.

To illustrate the preprocessing step, we will apply it to COBOL with embedded SQL instructions. The embedded SQL instructions that need to be analyzed to understand dataflow and control flow of a program are `select`, `insert`, `update`, `delete`, `declare cursor`, `open` and `fetch`.

The data exchange between the program and the DMS is done through *host variables*. The DMS uses *output host variables* to pass data and status information to the program. The program uses *input host variables* to pass data to the DMS.

To express the translation rules, some functions, that extract host variables and column name from SQL expressions, are needed:

- *input-couple(exp)*

  The list of the couples (`h-v`, `c`), such that `h-v` is an input host variable used in an SQL expression `exp` and `c` is the column associated with `h-v` in `exp`.
- *output-couple(exp)*

  The list of the couples (`h-v`, `c`), such that `h-v` is an output host variable used in an SQL expression `exp` and `c` is the column associated with `h-v` in `exp`.
- *host-var(list-couple)*

  The list of the host variables contained in `list-couple`. `list-couple` is a list of couples (`h_v`,`c`), where `h_v` is an input host variable and `c` is a column. `list-couple` is usually produced by *input_output(exp)* or *output_couple(exp)*.
- *column(list-couple)*

  The list of columns contained in `list-couple`. `list-couple` is a list of couples (`h_v`,`c`), such as `h_v` is an input host variable and `c` is a column. `list-couple` is usually produced by *input_output(exp)* or *output_couple(exp)*.
- *output-host-var(exp)*

  The list of all the output host variables used in `exp` and that do not appear in *host-var(output-couple(exp))*. This function is useful because there exist some output host variables that are not associated with a column, such as the status variables (telling if the query is correct or succeeds,...) or aggregation queries such as

  ```
  select count(x) into :nbr from customer
  ```

  the variable `nbr` is associated with no column.

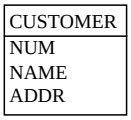


FIGURE 86. The `CUSTOMER` table.

To illustrate the translation process, the simple database schema of figure 86, will be used.

```
for each (h-v,c) in input-couple(query)
   generate(DIRECT-MAP h-v TO c)

generate(INDIRECT-MAP column(input-couple(query))
     TO column(output-couple(query)) ∪ output-host-var(query))

for each (h-v,c) in output-couple(query)
   generate(DIRECT-MAP c TO h-v)
```

FIGURE 87. The algorithm to translate a `select` query (`query`).

6.5.1. Select

A `select` query is translated in three parts. The first one represents the direct mappings between the input host variables and their corresponding columns. Those mappings appear in the `where` clause of the query. The second part is the indirect mapping from the columns associated with the input host variables (column(*input-couple(query)*)) to the columns associated with the output host variables and the output host variables not associated with a column (column(input-couple(*query*)) ∪ output-host-var(*query*)). The last part represents the direct mapping between the selected columns and their corresponding output host variables (*output-couple(query)*). These last mappings appear in the `select ... into` clause of the query. The algorithm to translate a `select` query is given in figure 87.

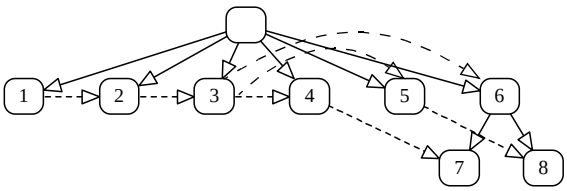
The translation may appear complicated but the resulting SDG has two interesting properties. The first one (that is the minimum requirement of the SDG) is that it is possible to compute a correct program slice for any instruction of the program. This slice contains the queries that influence the value of the variable for which the slice is computed and the slice also traverses the query, i.e., the slice contains also the instructions that influence the query. The second property of this SDG, is that the columns of the database that are used appear explicitly in the SDG. This is not necessary to compute a correct program slice, because the program does not know those columns since they are external to the program. But it is very important in the context of DBRE to know which column is linked to which host variable.

```
accept CUS-NUM.
exec sql
  select NAME, ADDR
  into :CUS-NAME, :CUS-ADDR
  from customer
  where NUM = :CUS-NUM
end-exec
if (SQLCODE = 0) THEN
  display CUS-NAME
  display CUS-ADDR.
```

a) The program fragment.

```
1  accept CUS-NUM.
2  direct-map CUS-NUM to NUM.
3  indirect-map NUM
      to NAME ADDR SQLCODE.
4  direct-map NAME to CUS-NAME.
5  direct-map ADDR to CUS-ADDR.
6  if (SQLCODE = 0) THEN
7    display CUS-NAME
8    display CUS-ADDR.
```

b) The translation.



c) The SDG.

FIGURE 88. A program fragment with a `select` query, its translation and SDG.

Figure 88.a is a fragment of a COBOL embedded SQL program, figure 88.b is the translation of the fragment and figure 88.c is the corresponding SDG. The bold lines of figure 88.b are the lines that belong to the slice computed with respect to the `display CUS-NAME` instruction.

```
for each (h-v,c) in input-couple(insert)
   generate(DIRECT-MAP h-v TO c)

generate(INDIRECT-MAP column(input-couple(insert))
      TO output-host-var(insert))
```

FIGURE 89. The algorithm to translate an `insert` instruction (`insert`).

6.5.2. Insert

An `insert` instruction can be easily translated as a direct mapping between each input host variable and its corresponding column and an indirect mapping between the columns and the status variables (`SQLCODE`). The algorithm to translate an `insert` instruction is given in figure 89.

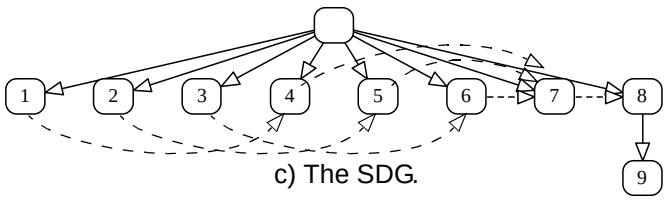
As in the `select` translation, if we only need a correct SDG to understand the program and we are not interested in its relation with the database, the translation can be simplified.

```
accept CUS-NUM.
accept CUS-NAME.
accept CUS-ADDR.
exec sql
  insert into CUSTOMER
    (NUM, NAME, ADDR)
  values (:CUS-NUM,
     :CUS-NAME, :CUS-ADDR)
end-exec
if (SQLCODE = 0) THEN
  go to ERR-INSERT.
```

a) The program fragment.

```
1  accept CUS-NUM.
2  accept CUS-NAME.
3  accept CUS-ADDR.
4  direct-map CUS-NUM to NUM.
5  direct-map CUS-NAME to NAME.
6  direct-map CUS-ADDR to ADDR.
7  indirect-map NUM NAME ADDR
      To SQLCODE.
8  if (SQLCODE = 0) THEN
9    go to ERR-INSERT.
```

b) The translation.



c) The SDG.

FIGURE 90. A program fragment with an `insert` instruction, its translation and SDG.

Figure 90.a is a fragment of a COBOL embedded SQL program that contains an `insert` instruction. Figure 90.b is its translation and figure 90.c the corresponding SDG.

```
for each (h-v,c) in input-couple(delete)
   generate(DIRECT-MAP h-v TO c)

generate(INDIRECT-MAP column(input-couple(delete))
      TO output-host-var(query))
```

FIGURE 91. The algorithm to translate a `delete` instruction (`delete`).

6.5.3. Delete

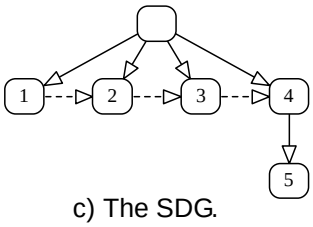
A `delete` instruction can be translated as a direct mapping between each input host variable and its corresponding column that appears in the `where` clause and an indirect mapping between the columns and the status variable (`SQLCODE`). The algorithm to translate a `delete` instruction is given in figure 91.

```
accept CUS-NUM.
exec sql
  delete CUSTOMER
    where NUM = :CUS-NUM
end-exec
if (SQLCODE = 0) THEN
  go to ERR-DELETE.
```

a) The program fragment.

```
1  accept CUS-NUM.
2  direct-map CUS-NUM to NUM.
3  indirect-map NUM
      to SQLCODE.
4  if (SQLCODE = 0) THEN
5    go to ERR-DELETE.
```

b) The translation.



c) The SDG.

FIGURE 92. A program fragment with a `delete` instruction, its translation and SDG.

Figure 92.a is a fragment of a COBOL embedded SQL program that contains a `delete` instruction. Figure 92.b is its translation and figure 92.c the corresponding SDG.

```
for each (h-v,c) in input-couple(update)
   generate(DIRECT-MAP h-v TO c)

generate(INDIRECT-MAP column(input-couple(update))
      TO output-host-var(update))
```

FIGURE 93. The algorithm to translate an `update` instruction (`update`).

6.5.4. Update

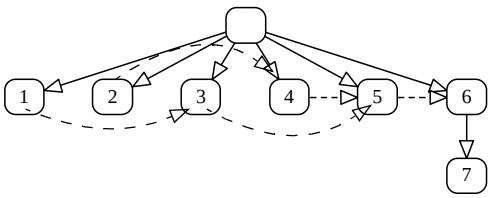
An `update` query can be translated in two parts. The first one represents the direct mapping between the input host variables and their corresponding columns. This mapping appears in the `where` clause and in the `update` clause. The second is the mapping between the columns associated with the input host variables (*column(input-couple(update))*) and the output host variables not associated with a column (*output-host-var(update)* = `SQLCODE`). The algorithm to translate an `update` query is given in figure 93.

```
accept CUS-NUM.
accept CUS-NAME.
exec sql
  update CUSTOMER
    set NAME = :CUS-NAME
    where NUM = :CUS-NUM
end-exec
if (SQLCODE = 0) THEN
  go to ERR-UPDATE.
```

a) The program fragment.

```
1  accept CUS-NUM.
2  accept CUS-NAME.
3  direct-map CUS-NUM to NUM.
4  direct-map CUS-NAME to NAME.
5  indirect-map NUM NAME
     to SQLCODE.
6  if (SQLCODE = 0) THEN
7    go to ERR-UPDATE.
```

b) The translation.



c) The SDG.

FIGURE 94. A program fragment with an `update` instruction, its translation and SDG.

Figure 94.a is a fragment of a COBOL embedded SQL program that contains an `update` instruction. Figure 94.b is its translation and figure 94.c the corresponding SDG.

6.5.5. Cursor

The difficulty of the cursor translation is that there are four instructions (`declare cursor`, `open`, `fetch` and `close`) that cannot be translated individually. The translation of the `declare cursor` instruction does not produce any mapping (or SDG vertices). But it declares the input host-variables and the columns that will be used by the other instructions (`open`, `fetch`). The `open` instruction is translated as the mapping between the input host variables and the corresponding column as declared in the cursor. The `fetch` instruction is translated by the mapping between the selected columns, defined in the cursor and the output host variables of the `into` clause of the fetch instruction. And finally, the `close` is not translated because it does not generate any mapping.

```
for each (h-v,c) in
          input-couple(declare_cur)
   generate(DIRECT-MAP h-v TO c)

generate(INDIRECT-MAP
     column(input-couple(declare_cur))
  TO output-host-var(declare_cur))
```

a) The algorithm to translate the `open` instruction.

```
generate(INDIRECT-MAP
     column(input-couple(declare_cur))
  TO column(output-couple(declare_cur
                          ∪ fetch)
      ∪ output-host-var(fetch))

for each (h-v,c) in
   output-couple(declare_cur ∪ fetch)
 generate(DIRECT-MAP h-v TO c)
```

b) The algorithm to translate the `fetch` instruction.

FIGURE 95. The algorithm to translate the `open` and `fetch` instructions that use the cursor declared by `declare-cur` instruction.

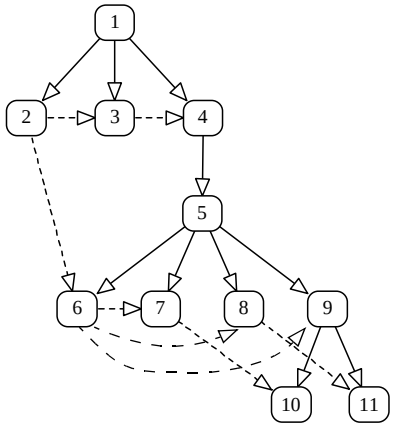
Figure 95.a is the algorithm that translates an `open` instruction and figure 95.b is the algorithm that translates a `fetch` instruction.

```
  exec sql
    declare C_ORD cursor for
    select NUM, ADDR
    from CUSTOMER
    where NAME = :CUS-NAME
  end-exec.
LIST-ORD.
  exec sql
    open C_ORD end-exec.
  perform READ-ORD
     until SQLCODE NOT = 0.
  exec sql
   close C_ORD_SEQ end-exec.

READ-ORD.
  exec sql
    fetch C_ORD
    into :CUS-NUM, :CUS-ADDR
  end-exec.
  if SQLCODE = 0
    display CUS-NUM
    display CUS-ADDR.
```

a) The program fragment

```
1  LIST-ORD.
2   direct-map CUS-NAME to NAME.
3   indirect-map NAME to SQLCODE.
4   perform READ-ORD
      until SQLCODE NOT = 0.

5  READ-ORD.
6   indirect-map NAME
      to NUM ADDR SQLCODE.
7   direct-map NUM to CUS-NUM.
8   direct-map ADDR to CUS-ADDR.
9   if SQLCODE = 0
10    display CUS-NUM
11    display CUS-ADDR.
```

b) The translation.



c) The SDG.

FIGURE 96. A program fragment with cursor manipulation instructions, its translation and SDG.

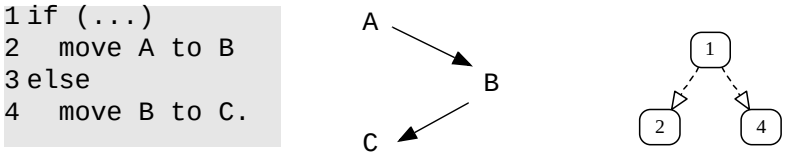
6.6. Other SDG analysis / usage

In the previous sections, program slicing was presented as a SDG traversal. In this section other SDG querying (traversal or visualization) will be presented.

The SDG is a representation of the program that contains all the control flows, dataflows and variable usages of the program. Program slicing is only one way to query it, but many other usages of this program representation can be considered. The SDG is easier to query than the program itself because it is an abstract representation stored as a graph and it is almost independent from the programming language. Therefore, many of the graph traversal techniques can be used. Because it

is language independent, once querying techniques and tools have been developed, they can be reused for other languages. The first SDG querying is some variant of the program slicing traversal restricted to data dependency arcs.

A *dataflow program slicing* can be defined as the program slicing where only the data dependency (data dependency, parameter-in, parameter-out) edges are used (the control edges are discarded). If a dataflow program slice, computed with respect to the instruction A, contains the instruction B, it means that there is a dataflow between some parts of variable of B to some parts of variable of A. This slicing technique gives incomplete slices, because it does not follow control edges. But it can be useful to find which variables are assigned to the slicing criterion variable to solve structure hiding problems such as decomposition of fields, anonymous fields and procedurally controlled foreign keys.



a) The program fragment. b) The VDG. c) The SDG.

FIGURE 97. The variable dependency graph and the SDG of a program fragment.

This querying technique gives more precise results than the variable dependency graph. The latter is constructed by an analysis of each instruction independently of the other one and does not take into account the control flow between instructions. The SDG construction uses the control flow of the program and thus, if there is no execution path between two instructions, it will never create a dataflow arc between them. In figure 97, the variable dependency graph shows that there is a dataflow between A and C (through B), but this dataflow is impossible because each instruction is in a different branch of the test and it is impossible to execute both instruction sequentially. On the other hand, the SDG does not discover this dataflow, because it is aware that both instructions are in different branches of the test.

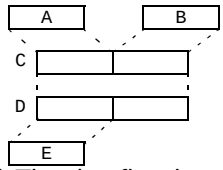
A data dependency edge from vertex *v* to vertex *w* means that there is a variable defined in *v* that is referenced by *w*. If an instruction contains more that one variable (or a compound one), it is impossible to know (in the SDG as defined until now) which variables influence the dataflow. So, if we follow the data dependency edges, it is possible to reach some vertex from which there is no dataflow to the slicing criteria.

```
01 A pic X(10).
01 B pic X(5).
01 C.
  02 C1 pic X(10).
  02 C2 pic X(5).
01 D.
  02 D1 pic X(10).
  02 D2 pic X(10).
01 E pic X(10).
```

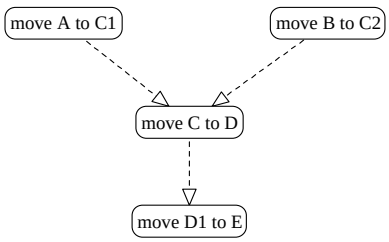
a) The variables declaration.

```
move A to C1.
move B to C2.
move C to D.
move D1 to E.
```

b) The program fragment.



c) The dataflow between variables.



d) The SDG without control flow.

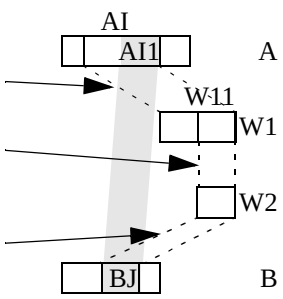
FIGURE 98. A program and its corresponding SDG.

For example, the dataflow slice of figure 98, with respect to the instruction `move D1 to E` and variable `D1`, contains all the program lines. But there is no dataflow nor control flow from `move B to C2` to the slicing criteria. This is because `move D1 to E` only references the first 10 characters of `D` (`D1`). `move C to D` defines all the characters of `D` and references all the characters of `C`. The first 10 characters of `C` are defined in `move A to C1` and the last 5 in `move B to C2`. In the instruction `move C to D`, the first character of `C` is assigned to the first character of `D`, the second character of `C` is assigned to the second character of `D`, etc. Thus there is no dataflow between `move B to C2` and `move D1 to E`, but the SDG only stores the variable defined and referenced by a vertex and not which referenced variable influence (dataflow) which of the defined variables.

To increase the precision of this technique, each instruction can be analyzed to determine which parts of the variable are used. This technique is called *dataflow program slicing with variable follow-up*. To compute this result, a path in the data dependency graph is chosen and this path is walked backward. For each vertex, on the path, a new parameter, *loc_ref*, is added. *loc_ref* is the list of the variables referenced by the current vertex and that influence the slicing criteria, i.e. there is a dataflow between *loc_ref* and the slicing criteria variable. *loc_ref* is computed recursively starting from the slicing criteria. *loc_ref* of the slicing criteria vertex (the last one) is initialized with the slicing criterion variables. To compute n_i.*loc_ref* for the previous vertex (with n_i the current vertex), we compute the intersection between n_i.*loc_ref* and n_{i-1}.*def* and substitute the variables with the corresponding one of n_{i-1}.*ref*.

```
fd A.
01 REC-A
  03 A1 X(20).
  03 AI.
    04 AI1 X(10).
    04 AI2 X(10).
fd B.
01 REC-B.
  03 B1 X(15).
  03 BJ X(10).
01 W1.
  02 W10 X(10).
  02 W11 X(10).
01 W2 X(10).
```

```
0Main.
1 read A.
  ...
2 move AI to W1.
  ...
3 move W11 to W2.
4 move W10 to W2.
  ...
5 move W2 to BJ.
  ...
6 write B.
```



1.loc_ref: `A[31-41]`

2.loc_ref: `A[31-41]`

3.loc_ref: `W1[11-20]`

5.loc_ref: `W2[1-10]`

6.loc_ref: `B[1-25]`

a) The program. b) The dataflow. c) *loc_ref.*

FIGURE 99. Attribute dependency detection using dataflow program slicing.

Figure 99 shows an example of this dataflow program slicing with variable follow-up. This technique gives very precise results with no noise, but there is some silence because it does not use the control flow edges of the SDG.

This technique works very well when there is a direct mapping between the referenced variables and the defined variable, as in `move A to B` instruction, where the first byte of `A` is assigned to the first byte of `B` and so on. However with the instruction `compute A = B * C`, it is very difficult, or impossible, to know which part of `B` and `C` goes into which part of `A`. Instructions such as `compute` are called *indirect mapping* instructions. When a dataflow program slice with variable follow-up is computed and it goes through a vertex that represents an indirect mapping. *C*ompute *loc_ref* as the variable referenced by the current vertex and the slicing criteria is impossible for a indirect mapping, because it is not possible to know which part ofthe referenced variable influence which part of the defined variable. So for an indirect mapping, *loc_ref* receives the list of the variables referenced in the vertex

```
0 new-order4.
1   display "customer number".
2   accept WO-ORD-CUS.
3   move WO-ORD-CUS to CUS-ID.
4   read CUS key CUS-ID
5     invalid key move 0 to FIND-CUS
6     not invalid key move 1 to FIND-CUS.
```

```
7   if FIND-CUS
8     move WO-ORD-CUS to ORD-CUS
9 .    display "order number"
10.    accept ORD-ID
     ....
11     write ORDER
    end-if.
```

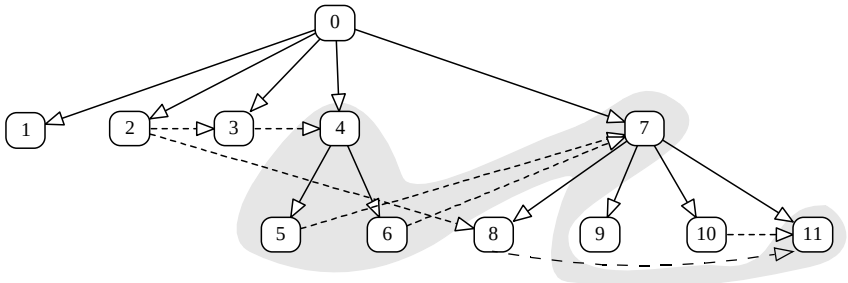


FIGURE 100. A paragraph that validates a foreign key and its SDG.

The analysis of the data dependency edges alone is sometimes insufficient to understand a program and to discover constraints. For example, the code of figure 100 implements a referential constraint between `ORD` and `CUS`, but if only the data dependency edges of the SDG are analyzed, the foreign key is not discovered. The value of `ORD-CUS` does not take its value from `CUS` but from `WO-ORD-CUS`. But if the control dependency edges are also used, as in classical program slicing with respect to line `11` and `ORD-CUS`, instruction `11` depends on 8, which depends on `7`, through a control edge. Instruction `7` receives its value from `5` or 6 that are control dependent form `4` (`read CUS`). So through this path that uses control and data dependency edge the foreign key can be discovered.

To extend the dataflow program slicing with variable follow-up to the usage of control dependency edges, the same rules are used as for the data dependency edges and when a control dependency edge is traversed backward, the *loc_ref* receives the list of the variables referenced in the vertex.

Until now all different SDG walk through algorithms presented are pre-defined algorithm (built-in a tool). It can also be useful to offer some kind of SDG querying language or SDG visualization tools to help the analyst to analyze the SDG. Such querying languages must offer primitives to select the vertices that define or reference a given variable, to select all the vertices that can be reached from the current one by following some kind of edges, to select vertices of a given type, etc. The visualization of the SDG is very difficult because of the large number of vertices and edges, so a realistic SDG visualization must only display some selected (with the querying language) vertices and edges.

6.7. Type inference

The basic idea of type inference ([Moonen - 2002]) is simple: if the value of a variable is assigned or compared to another variable, we want to infer that these two variables should have the same type. A type can play a number of roles: indicating the set of values that is allowed for a variable; grouping variables that represent the same kind of entities; hiding the actual representation used; providing a signature of a procedure.

The COBOL language does not support the notion of types. It is not possible to separate type definitions from variable declarations. When two variables need the same structure, this structure is repeated. The size of the data division makes difficult to determine if a structure repetition is accidental or whether it is intentional. Finally the absence of explicit types leads to lack of abstraction, since there is no way to hide the actual representation of a variable into same type name.

In DBRE, type inference can be useful to find new decompositions of attributes, data dependencies between two (groups of) attributes. If an attribute is of a given type and this type has a more precise decomposition, it is possible to refine this attribute such as the type definition. If two attributes are of same type, this is a hint that there is a potential dependency between these attributes.

To be able to analyze the types of the variables it is necessary to infer (automatically) the type of the variables. Moneen ([Moonen - 2002]) describes how type relation can be derived from the statements in a single COBOL program, and how this approach can be extended to system-level analysis leading to inter-program dependencies.

He defines three primitive types: 1) elementary types such as numeric values or strings; 2) arrays; 3) records. Initially every declared variable gets a unique primary type.

By looking at the expressions occurring in statements, an equivalence relation between primitive types can be inferred.

By looking at the assignments, a subtype relation can be inferred between primitive types. From an assignment of the form `move u to v`, it can be inferred that the type of `u` is a subtype of the type of `v`.

In addition to inferring type relations within individual programs, type relations can be derived at the system-wide level. The types of the actual parameters of a program call (listed in the `USING` clause) are subtypes of the formal parameters (listed in the `LINKAGE` section) and that variables read form or written to the same file or table have equivalence types.

6.8. Graphical visualization of the program

To perform program understanding, the analyst needs tools to easily represent and manipulate graphs. The graphs can be used to represent many different kinds of information, such as:

- Inter-program call graph: the vertices represent programs and the edges the call between two programs.
- Intra-program call graph: represents the call relations between the procedures inside a program (*perform* in COBOL).
- Database usage graph: the vertices are of two types (the programs and the collections/entity types) and the edges link the programs with the collections/entity types they use. The edges are labeled according to their usage (input, output or update).
- JCL graph: represents the chain of execution of the different programs with the files they use. Vertices are programs or files and the edges represent program chains and file usage.
- Hyperlink graph: the vertices represent documents and the edges the hyperlinks between the documents. This kind of graph can be useful for the analysis of Web documents.

This list of graphs is far from being complete and some of them can be combined, as the call graph and database usage. Some projects may need some particular graphs. The analyst needs flexible techniques and tools that allow representing almost any kind of graphs [van Deursen et al.-1998]. Those graphs can have different kinds of vertices and of edges. Those graphs are usually huge and complex. For example a call graph with 500 vertices is not exceptional. So the analyst need drawing and analysis help.

The drawing tools must offer functions to help laying out the graph such as minimizing the number of edges crossing, sorting the vertex by level, vertices alignments. But they must also offer some display functions as to color the vertices and the edges, to annotate them (visible annotations or invisible that can be used by further functions). Navigation functions are useful in big schemas to go from one vertex to the other through an edge.

Besides drawing tools the analyst needs analysis tools. For small graphs, the analyst can easily discover visually the graph's property, i.e. count the number of vertices, how many procedures call

a given one, find all the vertices that are reachable directly or indirectly from a given one, whether there exist some disconnected sub-graphs, etc. But in large graphs, he needs tools such as some statistical functions, functions to marks the vertices reachable directly or indirectly from a given one, functions to analyze or search vertices and edges of a given type.

Graphical visualization of the program can give very important hints about the structure of the program. But for large systems with several hundreds of programs the analyst can be flooded by the size and the complexity of these graphs. He needs help (tools) to manipulate and query these graphs.

CHAPTER 7 *Using program understanding in DBRE*

All the program understanding techniques presented in the previous chapter are general techniques, they are usually used by software engineers or maintainers to understand what a program is doing and how it is done. The scope of this thesis is not to understand the program itself but to use program understanding techniques to recover the implicit constraints that are enforced by the procedural part of the application.

This chapter explores how those program understanding techniques can be used to retrieve the implicit constraints and structures presented in section 4.3. For the most interesting constraints, the one who are generally searched for, it is explained how those techniques can be used and some hints are given on how the constraints discovery can be automated.

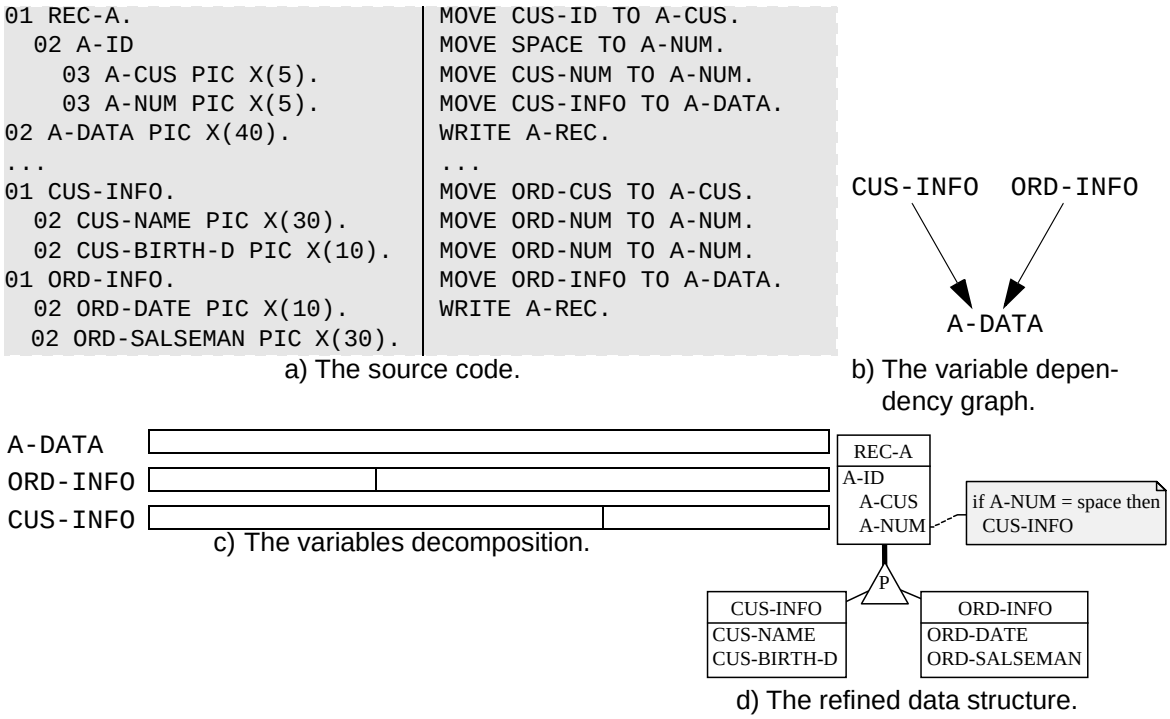


FIGURE 101. Example of incompatible attributes decomposition.

7.1. Fine-grained structure, attributes aggregation, anonymous attributes

Fine-grained structures, attributes aggregations and anonymous attributes are grouped in the same section because the same methods and techniques are used to detect them.

To discover a fine-grained structure using program understanding, the analyst has to find an attribute that has the same structure as a variable for which there exists a finer decomposition. Sometimes, the structure of both variables are incompatible, i.e. none of the structures is included in the other one. For example, in figure 101.b, there is a path in the variable dependency graph between `CUS-INFO` and `A-DATA` and thus the structure of `A-DATA` can be refined as `CUS-INFO` structure. There is another path in the variable dependency graph between `ORD-INFO` and `A-DATA` that has `A-DATA` as target. But the structure of `CUS-INFO` and `ORD-INFO` are incompatible. `ORD-INFO` is divided into 10+30 characters and `CUS-INFO` is divided into 30+10 characters (figure 101.c). There are at least two reasons for this incompatibility. The first one is that there is an error in the program. The second one is that the analyst misunderstood the program and needs to find another modelization for this data structure. In this example, the entity type `REC-A` is used to store two different kinds of information, the customer and the order. `REC-A` must be modeled as two different entity types (figure 101.d).

To discover attributes to be aggregated there must exist a variable in a program that is decomposed as the attribute. To refine (give a name and a decomposition) an anonymous attribute, a new decomposition of the parent (an attribute or an entity type) of the anonymous attribute is searched.

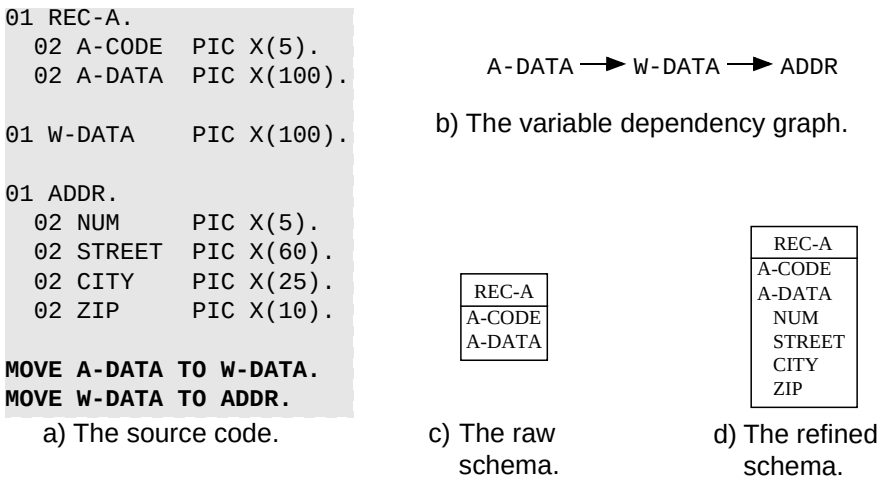


FIGURE 102. Example of attributes decomposition.

7.1.1. Variable dependency graph

Usually the relations used to construct the variable dependency graph are the assignment and the comparison operators. With these relations, the variable dependency graph can been seen as a simplified dataflow graph. The advantage of such variable dependency graph is that it is quite easy to compute, because a complete parser is not needed, only some instructions need to be parsed. The variable dependency graph is well appropriate to analyze some esoteric legacy language for which there does not exist a complete grammar and no parser is available. The drawback is that the vari-

able dependency graph is not a complete dataflow graph because some path in the graph can be infeasible during a correct execution of the program (see section 6.3).

If two variables are connected (directly or indirectly) in the variable dependency graph, then both variables are in relation, the analyst has to check if there exists a dataflow between these variables to determine if this is not noise generated by the variable dependency graph.

If the variables are in relation and the structure of the first one is finer than the structure of the other, the structure of the second one can be refined as the structure of the first one. For example, in the variable dependency graph of figure 102.b, computed with respect to the code of figure 102.a, `A-DATA` and `ADDR` are in relation and the structure of `ADDR` is finer than the one of `A-DATA`. Then the structure of `A-DATA` can be refined as the one of `ADDR` (figure 102.d).

7.1.2. System dependency graph

To know if there exists another decomposition for an attribute, the program slice with respect to this attribute (and its storage instruction) can be computed and analyzed to see if the attribute is in relation with a variable that has another structure. Classical program slicing with the control flow is not necessary, the dataflow program slicing is sufficient because only the variables that are connected to the selected attribute in the dataflow are of interest.

This can be easily automated by other SDG querying techniques that for each entity type storage instruction and each attribute, give the list of the variables (and their structure) from which there is a dataflow to or from the attribute.

7.2. Meaningful names

To find more meaningful names for entity types and attributes, the analyst needs to know the names of the variables that have different names, but have the same semantics (contains at some moment the same value). Another ways to find names is to look at the message displayed by the application or the comment in the source code.

He has to note the more meaningful name, but he must not change the attribute's name because during the data structure extraction process the logical schema must contains the original names; otherwise the programmer can have great difficulties to make the link between the current databases and programs and the logical schema.

This process is impossible to automate because the concept of meaningful is very subjective.

7.2.1. Variable dependency graph

If an attribute or an entity type is connected, directly or indirectly, in the variable dependency graph to variables, the analyst should decide that the name of one of the other variables is more meaningful than the attribute or entity type name.

7.2.2. System dependency graph

To find another name for an attribute (or an entity type), a program slice with respect to this attribute can be computed and analyzed to find a more meaningful name in the slice. Dataflow program slicing with variable follow-up is a good means to reduce the search space because only the variables assigned to the attribute are of interest and thus the control flow of the program is useless.

Program slicing can also be used to find in which context a variable is used by analyzing the messages displayed by the program slicing and the comments in the source code.

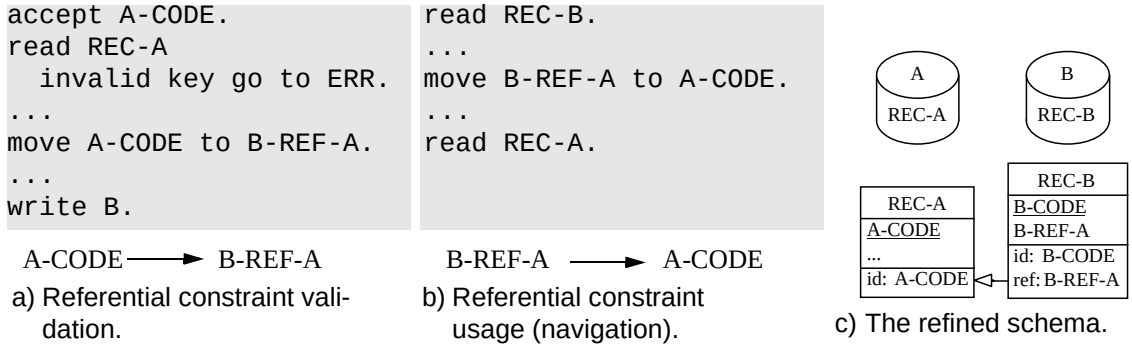


FIGURE 103. Two examples of procedurally verified referential constraint.

7.3. Referential constraints and data dependencies

Referential constraints can be seen as a special kind of data dependency. To find data dependencies, the analyst is searching for relations between two attributes (or groups of attributes).

There are two situations where a referential constraint can be detected. The first one is the referential constraint validation, where before the value of the referential constraint is stored (entity type modification, insertion or deletion) there must be some code that validates the value of the referential constraint (see figure 103.a). To validate the value of a referential attribute, the program reads the target entity type to check if there is a value of the target identifier. To detect such a referential constraint, a slice is computed with respect to the storage instruction and to validate the referential constraint, this slice must contains a read instruction of the target entity type.

The other situation where a referential constraint can be detected is during its usage, as in report generation (figure 103.b), because referential constraints can be viewed as a constraint but also as a navigation mechanism [Lopes et al.-2002]. To access the referenced entity type, the value of the referential attribute is used to access the referenced entity type through its identifier or in the other way, the value of the identifier of the first entity type is used as an access key to the second one, with respect to the referential attribute. To detect a referential constraint usage, a program slice is computed with respect to a read instruction and the attribute(s) that is used as an access key. This slice must contain another read instruction.

7.3.1. Variable dependency graph

Data dependencies and referential constraints can be discovered by the variable dependency graph if there is a path between attributes. Figure 103 shows two examples of variable dependency graphs with their respective code that can be used to detect a referential constraint. As suggested by this example the direction of the arcs in the graph does not influence the direction of the referential constraint (or the data dependency). If two nodes (variables) are connected (directly or indirectly), it means that there is a relation between these two nodes. The analyst is responsible to interpret the relation (referential constraint, data dependency).

The variable dependency graph generates noise and silence that the analyst has to deal with. The noise can be easily detected by further analysis of the schema structure, domain knowledge, data and program code. But as usual, the silence is difficult to detect. One way to reducing the danger of the silence is to understand its origin and which kinds of constraints are not discovered.

Variable dependency graph generates silence because it does not use the control flow and does not know the variable structure. The lack of control generates a lot of silence in the detection of referential constraint validations, because referential constraint validation relies a lot on control (to check if the referential attribute is present in the target entity type). But it produces less silence in referential constraint usage, when they are used to navigate in the database. So variable dependency graph can be a good choice to detect referential constraint in report generation modules.

There is less silence in data dependency detection because the target of the dependency is constructed from the origin, using mainly assignment (redundancy) and some computations. The main source of silence is the ignorance of the data structure of the variables.

7.3.2. System dependency graph

When a slice (with respect to a store or a read instruction) contains a read instruction, the probability to have a referential constraint is high, but the slice must be analyzed to determine between which attributes there is a referential constraint. It is up to the analyst to determine which attribute is used by the referential constraint. This validation is not difficult because the search space is reduced. He knows between which entity types there is a potential referential constraint and with the name, type and length of the attributes and domain knowledge he can easily discover the referential attribute.

Dataflow program slicing with variable follow-up can be used to know which attributes are in relation and thus gives more precise results to the analyst. It produces few silences for referential constraint usage because this usually only relies on dataflow. But it can produce silence in the case of referential constraint validation, because the validation often relies on control flow.

Data dependency as referential constraint is characterized by a relation (dataflow) between two entity types. The target of a referential constraint is an identifier of the entity type and data dependency is usually parallel to a referential constraint. They are parallel and in the same direction in the sense that if there is a data dependency between two entity types there must exists a referential constraint between these two entity types.

```
...
MOVE ORD-CUS-CODE TO CUS-CODE.
READ CUSTOMER
  INVALID KEY GO TO CUS-ERR.
MOVE CUS-NAME TO ORD-CUS-NAME.
...
WRITE ORD
  INVALID KEY ...
```

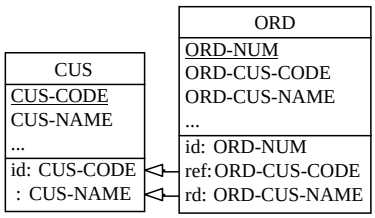


FIGURE 104. An example of data dependency.

For example in figure 104, there exists a referential constraint between `ORD-CUS-CODE` and `CUS-CODE` and a data dependency between `ORD-CUS-NAME` and `CUS-NAME`. The data dependency is parallel to the referential constraint because to fill the value of `ORD-CUS-NAME`, the program needs to know in which `CUS` to look.

As for the referential constraint, the slice with respect to a store instruction contains a read instruction, usually the same read instruction that is used to verify the referential constraint since the data dependency is parallel to the referential constraint. If program slicing is used, the slice has to be analyzed to know which attributes are in relation. So for the referential constraint, a more precise result can be obtained using program slicing with variable follow-up. If the identifier of the read entity type is copied into the stored entity type, there is a great chance that it is a referential constraint otherwise it can be a data dependency.

Usually data dependency can only be detected during insertion and modification of an entity type. Data dependency is an optimization technique used to save disk access, to prevent to read the other entity type. When the entity type is read, the other entity type referenced by the referential constraint does not need to be read.

```
FD ORDER.
01 ORD.
  ..
  02 ORD-DETAILS OCCURS 10.
    03 ORD-DET-PROD PIC X(3).
  ...
    03 ORD-DET-PRICE PIC 9(3).
FD PRODUCT.
01 PROD.
  02 PROD-CODE PIC X(3).
  ...
  02 PROD-PRICE PIC 9(3)
```

```
NEW-ORD.
  ...
  MOVE ORD-DET-PROD(IND) TO PROD-CODE.
  READ PRODUCT KEY IS PROD-CODE
    INVALID KEY GO TO PROD-ERR.
  MOVE PROD-PRICE TO ORD-DET-
     PRICE(IND).
  ...
  WRITE ORD
    INVALID KEY ...

CHANGE-PROD-PRICE.
  READ PRODUCT.
  ACCEPT PROD-PRICE.
  ....
  REWRITE PROD.
```

FIGURE 105. Example of non permanent data dependency.

Be aware of the conclusion, if program slicing detects that some attributes are in relation, it does not automatically mean that there is some data dependency. It can be some kind of business rule that can be documented but that is out the scope of the DBRE; it is more a part of the software reverse engineering. Sometimes data dependency is only verified at some moment, but it is not always verified. For example (figure 105), if `ORD` records the price of the ordered products (`ORD-DET-PRICE`), copied from the `PROD` entity type, but when the price of the product changes (in `CHANGE-PROD-`

PRICE paragraph), ORD-DET-PRICE is not changed, to know the price of the product at the order time. This is not a data dependency. This proves that all the data usage needs to be analyzed to be sure of a constraint and not only one usage!

7.4. Array set type, exact cardinality and attribute identifier

To find the set type and exact cardinality of a multivalued attributes the analyst needs to compute the program slice with respect to the store instruction and the multivalued attribute and then analyzes the slice to understand the algorithm used to fill the attribute. Does the user have to give at least one element or not (to find the exact minimal cardinality)? Does the user has to fill the entire attribute or can he stops when he wants (to find the exact maximum cardinality)? Is there an order or are there two elements with the same value in the attribute (find the set type)?

All this requires a lot of manual (and intellectual) work to understand the filling algorithm. There is no suggestion how to automate this process.

7.5. Identifier

To check an implicit identifier (attribute(s)' value is not yet present), the program has to read (sequentially) the existing instances of the entity type before it can write the new instance into the database to check. If the program slice is computed with respect to the write instruction and the whole entity type, the slice must contain a read instruction that reads the entity type itself. The analyst needs to check that the entity type is read to verify that the current value is not present in the database. This pattern (a write depending of a read of the same entity type) can also be the materialization of a recursive referential constraint.

7.6. Restricted domain

An attribute is of a restricted domain if the slice computed with respect to the store instruction and the attribute contains some tests that check the validity of the attribute's domain.

7.7. Embedded SQL

```
disp_ord.
  display "ord-num?".
  accept ORD-NUM.
  exec SQL
    select c.name, o.ord_date
    into :CUS-NAME, :ORD-DATE
    from c customer, d order
    where c.num = o.customer
          o.num = : ORD-NUM
  end-exec.
  if(SQLCODE = 0)
    display "Cust name : " CUS-NAME
    display "Order date : " ORD-DATE
    display "Price   Quantity"
    exec SQL
      declare cursor ord_detail
      select p.price, d.qty
      from p product, d detail
      where p.num = d.product
            d.order = :ORD-NUM
    end-exec
    exec SQL
      open ord_detail
    end-exec
    perform read-detail
      until SQLCODE not= 0
  end-if.

read-detail.
  exec SQL
    fetch ord_detail
    into :PROD-PRICE, :DET-QTY
  end-exec.
  display PROD-PRICE DET-QTY.
......
```

a) The original code fragment.

```
0  disp_ord.
1    display "ord-num?".
2    accept ORD-NUM.
3    direct-map ORD-NUM to o.num.
4    indirect-map o.num
          to c.name o.ord_date SQLCODE.
5    direct-map c.name to CUS-NAME.
6    direct-map o.ord_date to ORD-DATE.


7    if(SQLCODE = 0)
8      display "Cust name : " CUS-NAME
9      display "Order date : " ORD-DATE
10     display "Price   Quantity"









11     direct-map ORD-NUM to d.order
12     indirect-map d.order
             to p.price d.qty SLQCODE.
13     perform read-detail
         until SQLCODE not= 0
     end-if.

14 read-detail.
15   indirect-map to SQLCODE.
16   direct-map p.price to PROD-PRICE.
17   direct-map d.qty to DET-QTY.

18   display PROD-PRICE DET-QTY.
......
```

b) The transformed code fragment.

FIGURE 106. Example of program with embedded SQL.

In the previous chapter, an extension of the SDG has been presented to analyze the behavior of embedded code. This extended SDG, as presented, can be used to understand the program with embedded code but it is useless to analyze the embedded code itself.

For example, the analysis of the figure 107 SDG (that represent the program of figure 106) shows that the execution of the `fetch` instruction (lines 15-17) depends of the `SQLCODE` returned by the `select` (line 7) and thus the value of the `order.num` attribute (line 4). This is a hint that there is a referential constraint between `detail.order` and `order.num`. But the two other referential constraints between `order` and `customer` and between `detail` and `product` are not visible in the SDG because they are materialized by the embedded SQL queries (the `select` and the `cursor` declaration) and thus are not coded in the procedural part of the code.

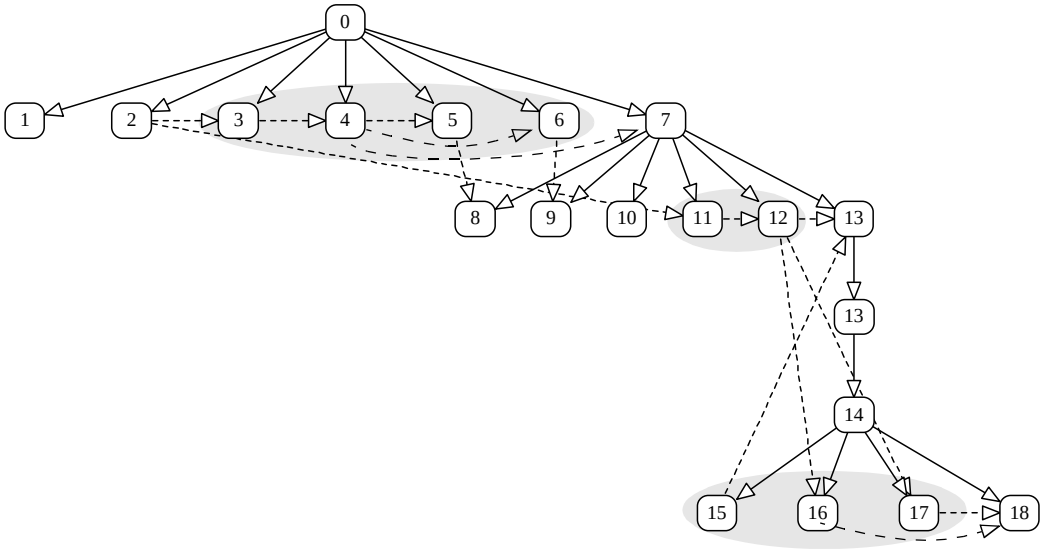


FIGURE 107. The SDG of figure 106.b source code.

To analyze embedded code, it is suggested to work in two steps. During the first one, each block of embedded code is analyzed individually [Petit-1996]. The purpose of this analysis is to extract from each embedded instruction the data structures and constraints implicitly implemented in this instruction. For example, if there is a join is an SQL query, then it can be interpreted as a referential constraint or a data dependency. This embedded code analysis need to be supported by tools such as tools to extract embedded code from the host language source code, pattern matching to find specific patterns (join, string manipulation, etc.) in the embedded code.

The second step is the analysis and the understanding of the program itself. To perform this analysis, classical SDG analysis is applied on the extended SDG. Only the host language program is analyzed without to worry about the embedded language. This is used, among other thing, to understand the links between the different embedded instructions.

7.8. Graphical visualization

Call graphs and database usage graphs can be very useful during project preparation [von Mayrhauser et al.-1993]. They allow the analyst to have an overview of the application, its components, and its complexity and to have a coarse idea of the time and budget needed to perform the DBRE. The analysis of those graphs is also a good way to know if the customer gives all DDL declarations and all the sources code indispensable to perform the DBRE.

During the data structure extraction call graphs and database usage graphs can be used to discover the general architecture of the application. This architecture helps the analyst in overall understanding of the application and to know which modules need to be analyzed (modules that modify the database) in priority.

The size and complexity of call graphs or usage graphs can be huge (several thousands of nodes and edges). This size and complexity prevent to display (or print) such graph. To overcome this, we

suggest to query the graphs to retrieve pertinent information. The query can produce some statistical results such as the number of node of a given type, the number of edges of a given type, the maximum number of edges per node, etc. Another possible result of a query is to extract a subgraph, such as all the nodes and vertices that can be reach from a given type, etc.

Graphs are a good medium to communicate with the customer and to generate reports.

CHAPTER 8 *CASE support*

Database reverse engineering appears as a demanding activity according to several dimensions: the size and variety of the information sources, the number and complexity of the elicitation techniques and the complexity of the target data structures. To perform efficiently a DBRE project, the analyst needs CASE tools to support his work. A CASE tool offering a rich toolset for reverse engineering is often called a CARE (*Computed-Aided Reverse Engineering*) tool. This tool can help the analyst during all the phases of a project and must be based on a common repository that stores all the information manipulated by the analyst (source code, schemas, etc.).

This chapter translates the characteristics of DBRE activities in CASE tool requirements and presents our implementation of a CASE tool that includes specific DBRE-oriented features.

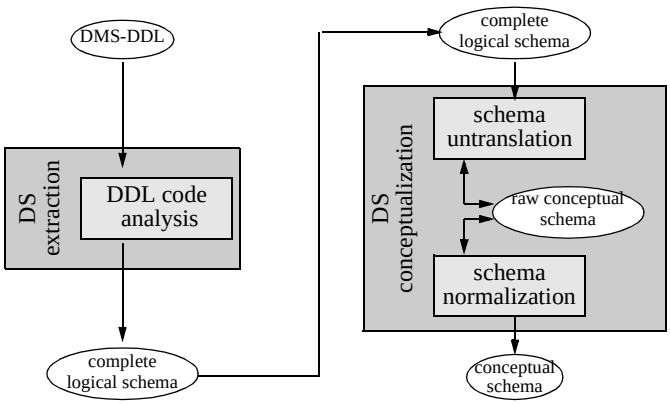


FIGURE 108. Simplified DBRE methodology proposed by most current CARE tools.

8.1. The limits of current CARE tools

It is interesting to use the methodology presented at the beginning of this thesis as a reference model against which existing methodologies can be compared, in particular, those used by the current CARE tools. Figure 108 summarized these methodologies as follows:

- *Data structure extraction*

  Most current CARE tools parse DMS-DLL schemas only (*DDL code analysis*). All the other sources are ignored and must be processed manually. For instance, these tools are unable to collect the multiple views of a COBOL application and to integrate them to produce the global COBOL schema. To minimize this drawback, most CARE tools only analyze some modern DMS-DDL such as various SQL dialects.

- *Data structure conceptualization*

  Current CARE tools focus mainly on untranslation (*schema untranslation*) and offer some restructuring facilities (*schema normalization*). These processes often are merged and are performed without any user intervention (fully automated). Therefore the analyst cannot drive the conceptualization process and he is provided with no tools to perform any transformation to the proposed schema. All performance-oriented constructs, as well as most non standard database structure (see [Blaha et al.-1995] and [Premerlani et al.-1993]) are completely beyond the scope of these tools.

8.2. Requirements

This section states some of the most important requirements an ideal DBRE support environment (a CARE tool) should meet. These requirements are the result of the analysis of the specific characteristics of the DBRE process and of the experience acquired during several DBRE projects of real size applications.

- *Flexibility*

  Observation: Reverse engineering activities differ from more standard engineering activities. Reverse engineering a database is basically an exploratory and often unstructured activity. Some important aspects of higher level specifications are discovered (sometimes by chance) and are not deterministically inferred from the operational specification.

  Requirements: The tool must allow the analyst to follow flexible working patterns, including unstructured ones. It should be methodology-neutral unlike forward engineering tools. In addition, it must be highly interactive.

- *Extensibility*

  Observation: Each project is different: new problems, new languages, new DBMS, new coding rules, etc. So each project requires specific reasoning and techniques. DBRE appears as a learning process.

  Requirements: Specific functions should be easy to develop, even for one-shot use. Existing functions must be easily integrated in new ones.

- *Source multiplicity*

  Observation: DBRE requires a great variety of information sources: DDL, data (from files, databases, spreadsheets,...), program sources code, program execution, program output, screens/ reports layout, CASE repository, documentation (paper and computer-based), interview, domain knowledge, etc.

  Requirements: The tool must include browsing and querying interfaces for these sources. Customizable functions for automatic and assisted specification extraction should be available for each of them.

- *Text analysis*

  Observation: DBRE requires browsing through huge amounts of text, and searching them for specific patterns, following static execution paths and dataflows and extracting program slices.

  Requirements: The CARE tool must provide sophisticated text analysis processes. The latter should be language independent, easy to customize and to program, and tightly coupled with the specification processing functions.

- *Program understanding*

  Observation: The code of the application is one of the major sources of information, since most implicit constraints must be implemented in the code to ensure the validity of the data.

  Requirements: The analyst heavily needs program understanding tools such as dataflow analysis, program slicing. Those tools could deal with very large program, legacy languages and must be easily customizable to other languages. The dilemma with the program understanding functions is that their precision depends on their dependency on the language. A language specific processor, such as program slicer, provides much better results than generic ones.

- *Name processing*

  Observation: Object names in the operational code are an important knowledge source. But these names often happen to be meaningless (e.g. `REC001-R018`) or at least less informative than expected (e.g. `INV-QTY`, `QOH`) due to the use of strict naming conventions. Many applications are multilingual[1], so data names may be expressed in several languages. In addition, multi-programmer development, long live time and maintenance, often induce non consistent naming. Names used in the programs and databases need to be used to name the objects in the physical schema to keep it synchronized with programs. During the conceptualization phase the physical names can be replaced by more meaningful ones.

  Requirements: The tool must includes sophisticated name analysis and processing functions. It must be also possible to keep the correspondence between the physical name of the objects and their more meaningful version in the conceptual schema.

- *Links with other CASE processes*

  Observation: DBRE is seldom an independent activity. For instance, forward engineering projects frequently include reverse engineering of some existing components; reverse engineering share important processes with forward engineering (e.g. conceptual normalization); reverse engineering is a major activity in broader processes such as migration, engineering and data administration.

  Requirements: A CARE tool must offer a large set of functions, including those that pertain to forward engineering.

- *Openness*

  Observation: There is (and probably will be) no available tool that can satisfy all corporate needs in application engineering. In addition, companies usually already make use of one or more CASE tools, software development environments, DBMS, 4GL.

  Requirements: A CARE tool must communicate easily with the other development tools, e.g. via querying hooks, communications with a common repository or by exchanging specifications through a common file format (XMI [XMI - 2002], GXL [Winter - 2002], XIML [Puerta et al.-2002]).

1. For instance, Belgium commonly uses three legal languages, namely Dutch, French and German. And English is often used by programmers as a common language.

- *Flexible specification model*

  Observation: As in any CAD activity, reverse engineering applies on incomplete and inconsistent specifications. However, one of its characteristics makes it intrinsically different from design processes: at any time, the current specifications may include components from different abstraction levels. For instance, a schema may include referential constraints as well as relationship-types.

  Requirements: The specification model must be wide-spectrum and provides artifacts for components of different abstraction levels.

- *Genericity*

  Observation: Tricks and implementation techniques specific to some data models have been found to be used in other data models as well (e.g. referential constraints are frequent in IMS and CODASYL databases). Therefore, many reverse engineering reasoning and techniques are common to the different data models used by current applications

  Requirements: The specification model and the basic techniques offered by the tools must be DMS-independent, and therefore highly generic.

- *Multiplicity of views*

  Observation: The specifications, whatever their abstraction level (e.g. physical, logical or conceptual), are most often huge and complex. Only one graphical view of the schema with some zoom-in and zoom-out is not enough to view and manipulate schemas. The analyst needs to examine and to browse through the information in several ways, according to the nature of the information he tries to obtain.

  Requirements: The CARE tool must provide several ways of viewing both source texts and abstract structures (schemas). Several different views (graphical and textual) are needed with powerful browsing functionalities and navigation functions (e.g., going from the origin of a foreign key to its target).

- *Rich transformation toolset*

  Observation: Actual database schemas may include constructs intended to represent conceptual structures and constraints in non standard ways and to meet non functional requirements (performance, distribution, modularity, access control, etc.). These constructs are obtained through schema restructuration techniques. They are discovered during the data structure extraction phase and need to be transformed during the data structure conceptualization phase.

  Requirements: The CARE tool must provide a rich set of schema transformation techniques. In particular, this set must include operators which can undo the transformation commonly used in practical database designs. The proposed transformations must preserve the semantics of the schema or at least warn the analyst when this semantics has changed.

- *Traceability*

  Observation: A DBRE project includes at least three sets of documents: the operational descriptions (e.g. DDL, source code), the logical schema and the conceptual schema. The forward and backward mapping between these specifications must be precisely recorded. The forward mapping specifies how each conceptual (or logical) construct has been implemented in the operational (or logical) specification, while the backward mapping indicates of which conceptual (or logical) construct each operational (or logical) construct is an implementation.

  Requirements: The repository of the CARE tool must record all the links between the schemas at the different levels of abstraction. More generally, the tool must ensure the traceability of the reverse engineering processes.

- *Automation*

  Observation: DBRE projects manipulate huge volume of information (source codes, texts) and the analyst needs to perform the same operation many times. The manual analysis is error prone.

  Requirements: The CARE tool must have powerful automation techniques. This automation can be built-in (i.e. functions that perform several of elementary actions) or available through some scripting function, so that the analyst can build his own tools.

8.3. The DB-MAIN CASE environment

DB-MAIN is a general purpose database CASE and meta-CASE environment that includes DBRE and program understanding tools. Its main goal is to support all the database application engineering processes, ranging from database development to system evolution migration and integration. Further detail on the whole approach can be found in [Hainaut et al.-1994].

The environment has been developed by the database engineering laboratory of the University of Namur (LIBD), Belgium, as part of the DB-MAIN project. As far as DBRE support is concerned, the DB-MAIN CASE tool has been designed to address as much as possible the requirements developed in the previous section. Extensions are being developed towards federated database methodology through the InterDB project [Thiran et al.-2000], data migration through Data Migration project [Delcroix et al.-2001] and methodological support for temporal database (TimeStamp project [Detienne et al.-2001]). More specifically it includes the following functions, components and capabilities:

- Classical functions to access, browse, create, update, copy, analyze and store the specifications (schemas and texts).
- Representation of the project history: processes, schemas, views, source texts, reports, generated programs and their relationships.
- A generic, wide-spectrum repository: the repository can store conceptual, logical and physical schemas and texts. It can represent entity relationship and UML models. Schema objects and text lines can be selected, marked, aligned, colored, copied and pasted.
- Semantic and technical annotations (text) can be attached to each specification object.
- Multiple views of the specifications (four hypertext and two graphical) with zoom-in and zoom-out. Some of the views are particularly intended for very large schemas.
- A tool box of about thirty semantics-preserving transformation operators which provide a systematic way to carry out such activities as conceptual normalization, or the development of optimized logical and physical schemas from conceptual schema and conversely (i.e. reverse engineering).
- The code generators generate the DDL code for such DMS as SQL, CODASYL, IMS and COBOL. There are three built-in SQL generators and an advanced one, written in *Voyager2*, generates checks, triggers and stored procedures to maintain additional constraints. XML-DTD and XML-schema generators have also been developed.
- Different report generators, from the simplest one that produces the same output as the current textual view, to a more sophisticate report in RTF with sophisticated page layout.
- Code parsers extracting physical schema from SQL, ODBC, COBOL, CODASYL, IMS DDL and XML-DTD.

- Text analysis tools such as pattern matching.
- Program understanding tools such as program slicing and variable dependency graph.
- Name processing to clean, normalize, convert or translate the names of selected objects.
- A history manager which records the engineering activities of the analyst and which makes their further replay possible.
- Import and export of specifications in a readable textual format.
- A series of assistants, which are expert modules in specific kinds of tasks or in classes of problems and which are intended to help the analyst in frequent, tedious or complex activities. It allows the analyst to develop scripts that automate frequent processes. Six assistants are available at present: global transformations (elementary and advanced), schema analysis, schema integration, text analysis and reference key analysis and discovery.
- Process modeling: specific methods can be defined and enforced by the tool. A method is defined by a MDL (*Method Definition Language*) script, compiled as a part of the repository, then enacted by the method engine [Roland et al.-2000].
- Extensibility: new functions, such as specific report and code generators, DDL analyzers or specifications checkers, can be developed in *Voyager2* [Englebert-2000]. This language allows the CASE engineer to develop new functions, which will be seamlessly incorporated in the tool without any modification of the tool kernel. *Voyager2* is a complete 4th-generation language that offers predicative access to the repository, easy analysis and generation of external texts, definition of recursive functions and procedures, a sophisticated list manager and direct access to the build-in functions (as the transformations). It makes the rapid development of complex functions possible.
- New properties can be dynamically added to the objects. Each type of object has built-in properties (e.g. attributes have a name, a type, a length, etc.). It is possible to add new properties to any type of object of the repository. For example, it is possible to add a property (*eng_name*) to the entity types that contains the name of the entity type in english.
- The behavior of the tools can be modified by the addition of pre- and post-processing functions (as triggers written in *Voyager2*) to the creation, deletion and transformation of the objects.

The remainder of this section will present in more detail the aspects and components of the DB-MAIN tool which are directly related to DBRE activities.

8.3.1. User interface

Besides fairly standard graphical user interface, DB-MAIN offers additional formats that can be useful for large schemas.

The tool allows the creation, modification, examination and analysis of the specifications. It must be able to process large schemas (e.g. 500 record types with 10000 fields) and texts (e.g. beyond 100000 LOC).

It quickly appeared that more than one way of viewing schemas is necessary. For instance, a graphical representation of a schema allows an easy detection of certain structural patterns (as N-ary relationship types) and manipulation of small to medium schemas. But positioning objects of large and complex schemas can prove difficult. It can take more than two hours to position the objects of a schema of about 300 entity types and 300 relationship types. A textual representation is better suited than the graphical one to analyze name correspondences and similarities and to browse

through large schemas. This is especially true in DBRE, where the schema can be very large and is extracted from the DDL, without any predefined graphical positions.

DB-MAIN currently offers three different kinds of schemas (extended entity relationship, UML class diagram and processing). The entity relationship and UML class diagrams represent data schema in their respective model. In the graphical and textual views objects can be marked and colored. Graphical views have zoom-in, zoom-out and alignments functions. The selected, marked and colored objects are kept from one view to the other.

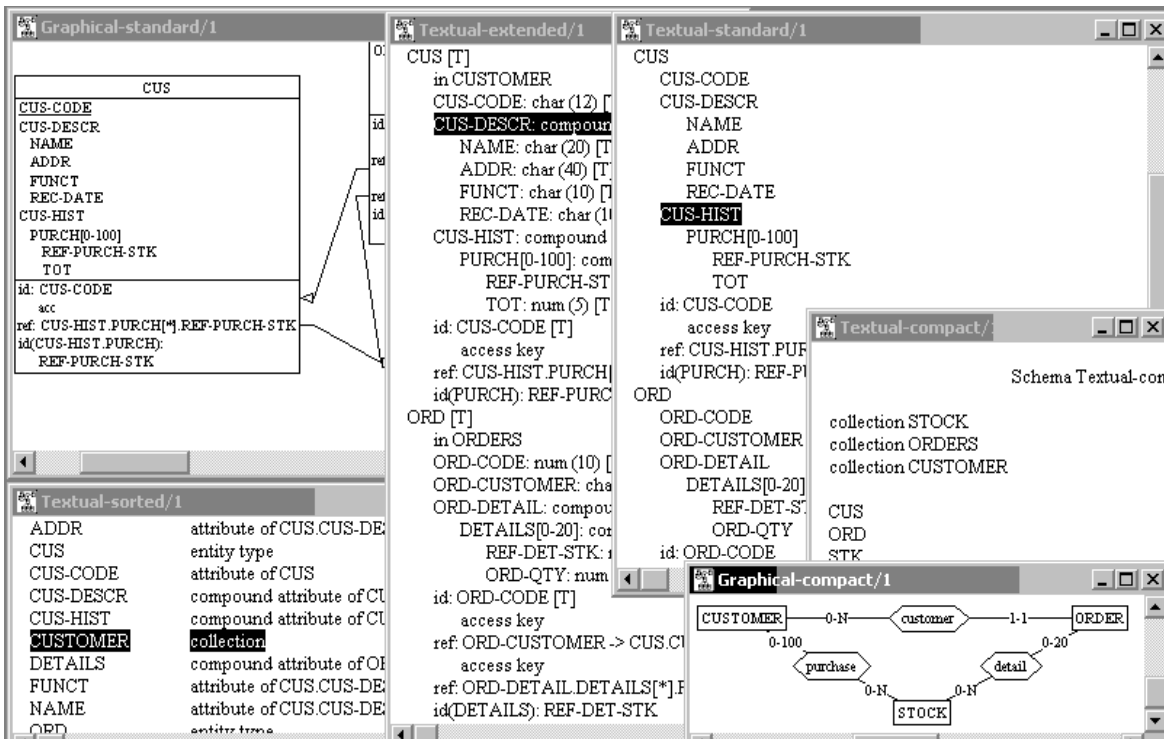


FIGURE 109. The six different extended entity-relationship views offers by DB-MAIN.

8.3.1.1. Entity relationship schema

Figure 109 presents the 6 views of an extended entity relationship schema (4 hypertext and 2 graphical).

The four extended entity-relationship textual views with hyperlinks are:

- *Textual-compact*: sorted list of entity types, relationship types and collections.
- *Textual-standard*: same as the compact one with attributes, roles, groups, processing units and is-a relations.
- *Textual-extended*: same as the standard one with the domain of the attributes and entity types - relationship types cross-references.
- *Textual-sorted*: sorted list of all the objects (entity types, relationship types and attributes) names.

The hypertext views provide an easy way to navigate through a schema by following the roles, foreign keys and is-a relations, i.e., going from an entity type to its neighbor relationship types and vise-versa.

The two extended entity-relationship graphical views are:

- *Graphical-compact*: graphical representation of the entity-types, relationship types, roles and collections.
- *Graphical-standard*: same as the compact one with the attributes, groups, processing units and groups. It is possible to customize the views by displaying or hiding the domain of the attributes, the attributes, the processing units and the groups.

8.3.1.2. Processing schema

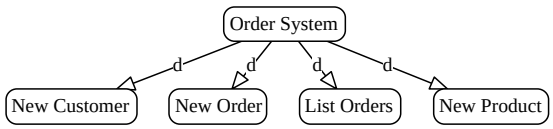
Processing schemas are used to represent processes. Three kinds of nodes are provided: processing unit, internal data objects and external data objects, as well as three kinds of relations: call, decomposition and in-out.

A *processing unit* describes any processing components of an application or of an information system. According to the level of abstraction at which the description has been developed, a processing unit can model a task, an organization function, an activity, a procedure, a program, a predicate, a trigger and even a mere statement.

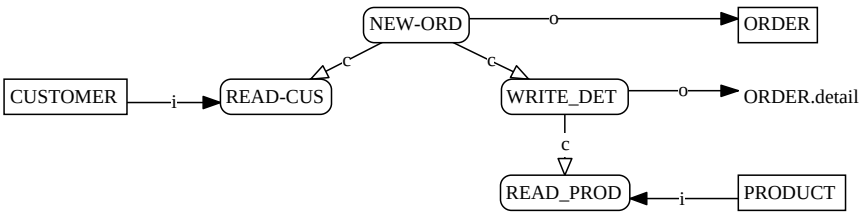
An *internal data object* can be a data type, a variable, a constant or any object known by the processing units of the schema but that is local to this schema. An internal object can be used as input or output of processing units.

A data object used in a processing schema that has been defined in a data schema is called an *external data object*, such as entity types, attributes, collections or relationship types. For instance, a procedure that reads *CUSTOMER* entities (described in a data schema) appears in a processing schema where *CUSTOMER* is declared external.

Relations describe how a processing unit relates to other processing units and to internal and external data objects. There are three kinds of relations.



a) Decomposition relations.



b) Call and in-out relations.

FIGURE 110. Example of the different relations in a processing schema.

A processing unit can be made up of several components which are themselves processing units (dotted lines figure 110.a), this relation is called *decomposition*.

The *call* relation states that a processing unit calls, or uses services from, other processing units (labelled "c" in figure 110.b).

A processing unit can use/read data objects and create/delete/update others, these relations are called *in-out* (labelled "i" for input; "o" for output; "u" for update in figure 110.b). Data objects can be internal (local to the schema) or external (defined in an other schema).

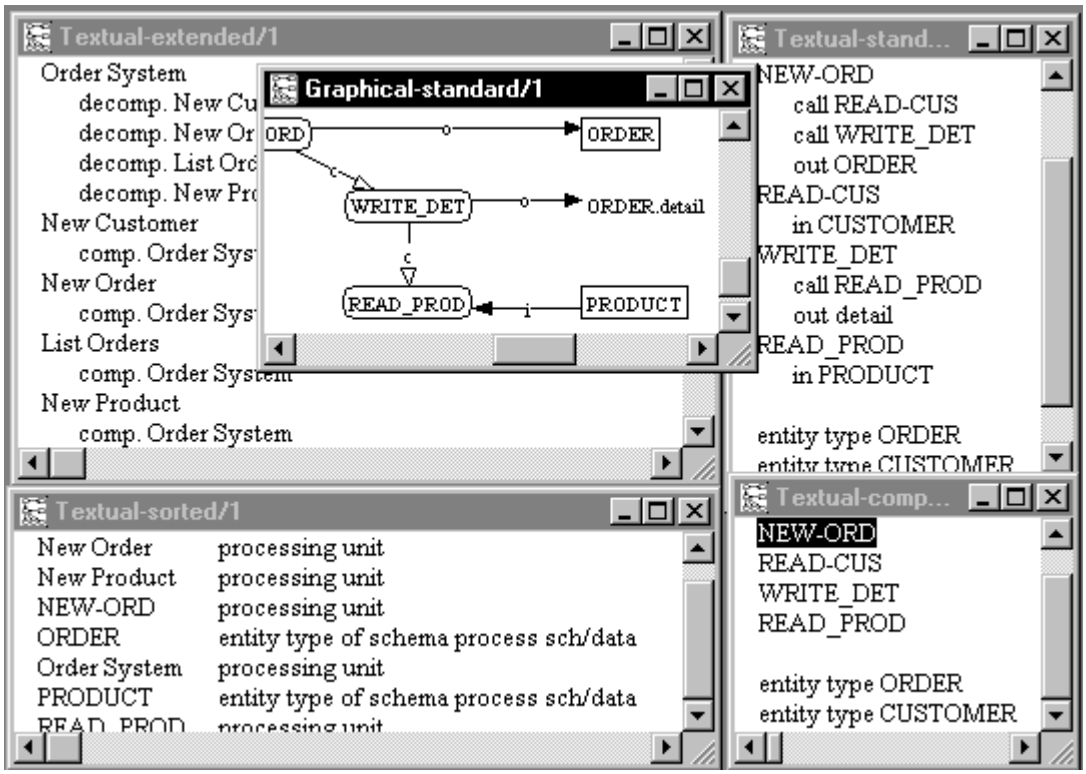


FIGURE 111. The five different processing views offers by DB-MAIN.

Figure 111 presents the five views of a processing schema:

- *Textual compact*: sorted list of processing units and data objects (internal and external).
- *Textual standard*: same as compact one with the relations between processing units and data objects.
- *Textual extended*: same as the standard one with relations cross-references.
- *Textual sorted*: sorted list of all the objects (processing units and data objects).
- *Standard graphical*: graphic representation of the processing units and data objects with the relations.

8.3.2. DDL extractors

The DB-MAIN CASE includes DDL extractors for popular DMS, such as COBOL, IMS, CODASYL, SQL, ODBC, Access and XML-DTD. Some of these extractors (COBOL, IMS, CODASYL,

SQL and ODBC) are built in the CASE tool kernel and other (Access and XML-DTD) are external modules written in *Voyager2*. For additional DMS, specific extractors can be developed in *Voyager2*.

These processors create an abstract schema expressing the physical concepts of the DDL text or of the data dictionary that declares the data structures. These extractors produce one schema per DDL analyzed or can store all the definitions in the same schema. If the extractors cannot interpret some part of the code, such as the body of the SQL triggers, they store them in the description of the corresponding object so that they can be analyzed by post processors.

8.3.3. Pattern matching

The simplest way to find some definite information in a program source code is to search the program source text for some patterns or clichés. DB-MAIN pattern matching can include wildcard, characters ranges, multiple structures, variables and can be based on other defined patterns.

For example, patterns can be defined to match any numeric constant, or the various kinds of COBOL assignment statement or some select-from-where SQL queries.

The DB-MAIN *pattern matching* function allows searching text files, object names or object descriptions for definite patterns expressed in a *Pattern Definition Language* (*PDL*). The left hand side of the pattern (left of the '`::=`') is the name of the pattern and the right hand side is its definition and is terminated by a '`;`'. The definition of the pattern can contain regular expressions (à la *grep*), wildcard, character ranges, multiple structures, variables (preceded by the `@` symbol) and other defined patterns. Pattern definitions cannot contain forward references, i.e,. all the patterns used in a pattern definition must be defined before.

```
- ::=   /g"[ /t/n]+";
var ::= /g"[a-zA-Z][-a-zA-Z0-9]*";
var_1 ::= var;
var_2 ::= var;
move ::= "move" - @var_1 - "to" - @var_2 ;
```

FIGURE 112. Patterns definition for detecting COBOL assignment. '`-`' designates any non-empty separator, '`var`' any alphanumeric string beginning with a letter (a variable name).

As an illustration, figure 112 is the definition of the pattern `move` that matches a COBOL assignment. The pattern '-' is defined as a regular expression (`/g"..."`) which represents one or more (`+`) separators (space, tabulation or newline). The pattern `var`, that represents a COBOL variable name, is defined as a regular expression which matches any alphanumeric string beginning with a letter. `var_1` and `var_2` are defined as `var`. The COBOL assignment, `move`, is defined as the string *"move"* followed by a non-empty separator (`-`), followed by a COBOL variable (`var_1`), followed by a non-empty separator (`-`), followed by the string *"to"*, followed by a non-empty separator (`-`), followed by a COBOL variable (`var_2`). The matching values of `var_1` and `var_2` are stored into two PDL variables named `var_1` and `var_2`. The value of those two variables can be used by other tools. For example, they can be passed as parameter to a *Voyager2* procedure.

The PDL variables can also be instantiated before the search takes place to reduce the search space. For example, if we do not want to find any assignment, but only those that assign a value to the variable `Cus-Name`, `var_2` can be instantiated to `Cus-Name` before the search takes place.

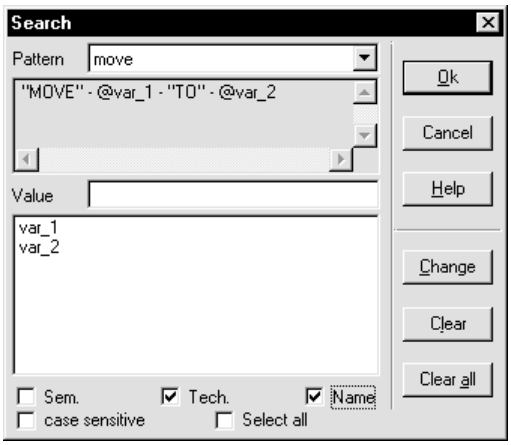


FIGURE 113. The search tool dialog box.

The search tool (figure 113) allows to select a pattern and to instantiate the variables. The search can take place in an external text or in a schema (name or description of the objects). This search tool is mainly used for visual inspection: it selects the next matching string or it can select all the matching strings in the product (select all).

```
select *
from CUTOMER, ORDER
where CUSTOMER.NAME = 'Dupont'
     and CUSTOMER.CUS-CODE
          = ORDER.ORD-CUST;
```

a) A SQL join.

```
! ::= any_but("where");
§ ::= any_but(";");
join ::=
    "from" ! (@T1 ! @T2| @T2 ! @T1)
    "where" § @T1 "." @C1 -
              = - @T2 "." @C2;
```

b) A pseudo-pattern to match a SQL join.

FIGURE 114. Example of pattern impossible to write in PDL.

The first experiments have quickly taught us that pattern-matching works fine for locally concentrated patterns, but can prove difficult to use for large patterns. It is not possible to write a pattern that contains an expression that matches any string not including an expression. For example, it is not possible to write a pattern that detects any SQL join. Figure 114.a shows a SQL join and figure 114.b shows a pseudo-pattern which would be necessary to detect if. The current pattern matching engine does not offer any way to express the "`any-but`" expression.

A Voyager2 procedure can be attached to a pattern in such a way that each instantiation of this pattern triggers the execution of the procedure. The procedure uses the PDL variables as input parameters. In this way, the analyst can build powerful custom tools that perform automatically some actions each time a pattern is detected (see annex A section A.2 for more detail).

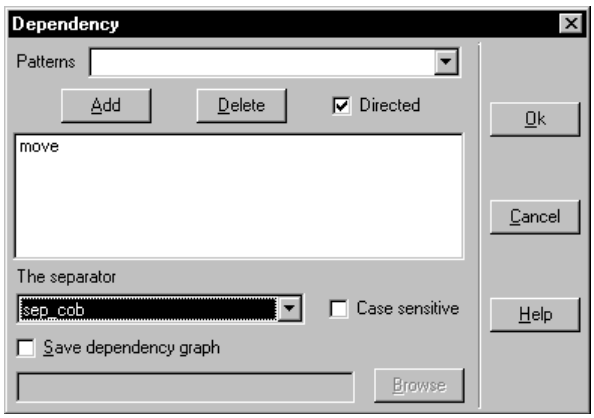


FIGURE 115. The variable dependency graph dialog box.

```
move ::= "move" - @var_1 - "to" - @var_2 ;
redefine ::= @var_1 - "redefines" - @var_2;
write ::= "write" - @var_2 - "from" - @var_1;
if ::= "if" - @var_1 - rel_op - @var_2;
```

FIGURE 116. Example of patterns used to compute the variable dependency graph in a COBOL program. Variables `@var_1` and `@var_2` define the nodes while the edges are built from the instantiation of the patterns.

8.3.4. Variable dependency graph

The *variable dependency graph* (see figure 115) tool builds a graph whose nodes are the variables of the program to be analyzed and the edges are relationship between these variables. These relationships are defined by selecting PDL patterns with two variables named `var_1` and `var_2`, the edges may be directed and if so, they are directed from `var_1` to `var_2`. For instance, figure 116 displays patterns that can be used to build a graph in which two nodes are linked if their corresponding variables appear simultaneously in a single assignment statement, in a redefinition declaration, in an indirect write statement or in comparisons.

This tool can be used to solve structure hiding problems such as the decomposition of attributes, anonymous attributes and procedurally controlled referential constraint.

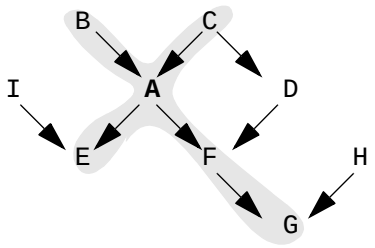


FIGURE 117. Example of node that are directly and indirectly connected to A.

Two visualization formats of the variable dependency graph are available. The first one is contextual. The analyst selects (clicks on the variable with the mouse's right button) a variable in the source code, in the declaration or in the procedural code, then all the occurrences of variables

connected, directly or indirectly, to the selected variable are colored in the source code. The nodes, that are connected indirectly to the selected node, are the nodes that can be reached from the selected node by following the edges forward and the nodes that can be reached from the selected node by following the edges backward. For example, figure 117 shows the nodes that can be reached directly and indirectly from A, namely B, C, E, F, G. Nodes I, D and H are not part of this set.

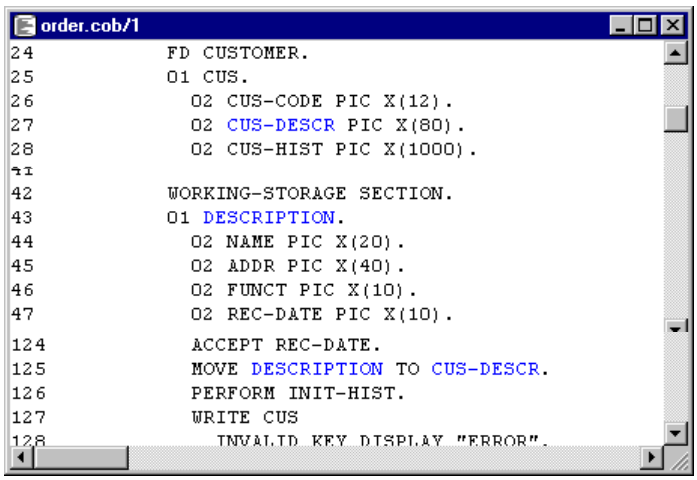


FIGURE 118. Example of variables belonging to the variable dependency graph shown in context.

With this visualization technique the analyst can observe the variables in their context, i.e., he can see the comments and the instructions that use the variables. But in large programs, he only has a partial view of all the variables connected.

The second view is graphical and represents the graph itself. In this view, it is easy to observe the cluster of variables in connection. The analyst has a global view of the dataflow of the program. This view, that does not belong to the kernel of DB-MAIN, can be built by saving the graph, then by using the processor `depend.oxo` to create the dependency graph as an entity-relationship schema (see annex A section A.4.3 for detail).

8.3.5. Program slicing

```
190NEW-ORD.                          211    INVALID KEY DISPLAY "ERROR".
191* new order input                 216READ-CUS-CODE.
192  DISPLAY "NEW ORDER".            217* order customer input
193  DISPLAY "ORDER NUMBER : "       218  DISPLAY "CUSTOMER NUMBER : "
194    WITH NO ADVANCING.            219    WITH NO ADVANCING.
195  ACCEPT ORD-CODE.                220  ACCEPT CUS-CODE.
196                                   221  MOVE 0 TO END-FILE.
197  MOVE 1 TO END-FILE.             222  READ CUSTOMER INVALID KEY
198  PERFORM READ-CUS-CODE           223    DISPLAY "NO SUCH CUSTOMER"
199          UNTIL END-FILE = 0.     224    MOVE 1 TO END-FILE
201 MOVE CUS-CODE TO ORD-CUSTOMER.   225  END-READ.
 ...
210  WRITE ORD
```

FIGURE 119. Example of program slice.

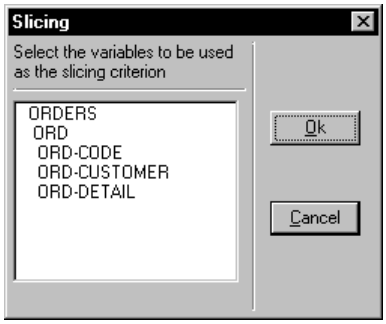


FIGURE 120. The dialog box used to select with respect to which variable the program slice must be computed.

DB-MAIN offers a program slicing tool for COBOL programs. The user selects an instruction and one or more variables referenced by the instruction. The program slicing tool identifies and colors the program slice with respect to the selected instruction and the list of variables. The tool colors the lines of the slice in the complete source code. This makes it possible to examine the slice in context. For example, figure 119 shows a part of a program that includes the program slice computed with respect to line 210 and `ORD-CUSTOMER,` shown in bold. The context of the slice makes it possible to analyze related information such as comments and error messages (lines 218 and 223).

To use the program slicing, the user selects a line in the source code and uses the **Assist / Text analysis / program slicing** command. The program slicing tool asks with respect to which variable the slice must be computed (figure 120) and then colors the lines bellowing to the slice.

Functions to mark, color and copy the lines of a slice are available to extract the program slice and to store it in a separate file.

This program slicing tool is very useful for visual inspection of the code, but it is not adapted for large projects with thousands of slices to compute and to analyze in hundreds of different source codes. So there exists a "command line" (Windows and Unix) version of the program slicing that can be used in shell scripts and run on a powerful machine. The command line has numerous options (see annex A, section A.5) to designate the input and output files and format, the starting lines and variables, etc.

For example, to detect data dependency the "command line" program slicing can be used to find all the variables that are referenced in a write instruction and that received their value from a variable defined in a read instruction, i.e., an attribute of the read entity type. To compute such slices for the `order.cob` program, the analyst can execute the following command.

```
slice -v -s write -c read -a var -o order.dep order.cob
```

The options of this command can be interpreted as follows. `-v` computes the variable follow-up program slicing. `-s write` computes the program slicing with respect to each `write` (or `rewrite`) instruction and its record. `-c read` checks if there is a `read` (or `start`) instruction in the slice. If the slice does not contain a `read` instruction, it is not memorized. `-a var` option is used to display (into the output file) the variables of the `read` and `write` instruction between which there is a dataflow. `-o order.dep` specifies that the output must be saved in the `order.dep` file.

```
begin : 208
end : 196
origin: CUSTOMER (1-12) key CUS-CODE
current  ORDERS (11-22) ORD-CUSTOMER
flow : 208,187,196
```

a) Dataflow only result.

```
begin: 226
end: 196
origin: LIST-DETAIL (1-5) REF-DET-STK
current: ORDERS (23-27) ORD-DETAIL
control: STOCK (1-5) key STK-CODE
control: EXIST-PROD (1-1)
flow: 226,227,228,245,194,196
```

b) Result with control flow.

FIGURE 121. Example of the output of the program command line program slicing tool.

```
208   READ CUSTOMER INVALID KEY ...
187   MOVE CUS-CODE TO ORD-CUSTOMER.
196   WRITE ORD...
```

FIGURE 122. Fragment of the program to understand the dependency between line 208 and 196.

An extract of the `order.dep` file is shown in figure 121. The first part (figure 121.a) can be interpreted as the existence of a program slice with respect to line 196 (`WRITE ORD`) that contains a read instruction (line 208) and there is a dataflow between `CUS-CODE` (origin) and `ORD-CUSTOMER` (current). The instructions used to detect this dataflow are given by the flow line and are displayed in figure 122.

```
FD ORDERS.                                  225   MOVE PROD-CODE TO STK-CODE.
  01 ORD.                                   226   READ STOCK INVALID KEY
    02 ORD-CODE PIC 9(10).                  227     MOVE 0 TO EXIST-PROD.
    02 ORD-CUSTOMER PIC X(12).              228   IF EXIST-PROD = 0
    02 ORD-DETAIL PIC X(200).               230   ELSE
                                            231     PERFORM   ...
01 LIST-DETAIL.                             245   MOVE PROD-CODE
  02 DETAILS OCCURS 20 INDEXED BY IND.                   TO REF-DET-STK(IND-DET)
    03 REF-DET-STK PIC 9(5).                194   MOVE LIST-DETAIL TO ORD-DETAIL.
    03 ORD-QTY PIC 9(5).                     196   WRITE ORD
01 EXIST-PROD PIC 9.
```

FIGURE 123. Fragment of the program to understand the dependency between the line 226 and 196.

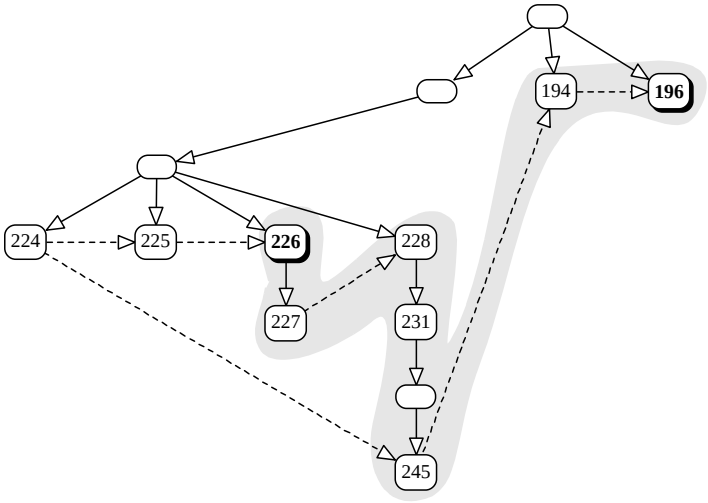


FIGURE 124. The SDG of the figure 123.

The second part (figure 121.b) is more difficult to understand because fully understanding the relation between the `read` instruction (line 226) and the `write` instruction (line 196) requires not only dataflow analysis but also control flow analysis. The instructions used to detect this dependency are displayed in figure 123. To find this dependency, the program slicing uses two control flow edges (see figure 124). The first one goes from line 226 to 227: the `read` (line 226) can be seen as a test, to test if the record key is valid or not. The second one goes from line 228 to 231 to 245: the `else` part of the `if` statement is a `perform` (line 231) that calls `UPDATE-ORD-DETAIL` in which `REF-DET-STK` receives its value. The two lines beginning by control in figure 121.b give the variables that are used in the two tests (line 226, 228). The `current` line gives the attribute of the `ORDER` entity type that receives the value. It is not the whole attribute `ORD-DETAIL` but only a part of it, from byte 28 to 32 relatively to the beginning of `ORDER`.

This result shows that there is a dependency between the value of `STK-CODE` (the primary key of `STOCK`) and a part of `ORDER` but this is not a dataflow. The analyst has to analyze manually the program to understand the dependency. In this example, at line 225 (not part of the slice between line 226 and 196) `STK-CODE` receives its value from `PROD-CODE` and if `STK-CODE` (so `PROD-CODE`) is an existing value then `PROD-CODE` is moved to `REF-DET-STK` (line 245) and `REF-DET-STK` is moved to `ORDER` (line 194). This suggests that there is a referential constraint between `STK-CODE` and `REF-DET-STK`.

For real size project, as in this example, the interpretation of the result of variable follow-up program slicing that contains control dependencies is not easy. The interpretation of variable follow-up program slicing with only data dependency is straightforward. Most of the foreign key can be discovered by only analyzing the variable follow-up program slicing computed with data dependency.

8.3.6. Referential key assistant

FIGURE 125. Referential constraint assistant dialog (manager component).

Referential constraints are one of the most important structures to elicit in almost every DBRE projects. A dedicated analyzer helps the analyst to discover potential referential constraints by the analysis of the database schema. The *referential key assistant* implements the most common heuristics. This assistant is divided in two components, namely, the constraint manager and the search engine. The first one (figure 125) is the component manager; it is used to give the search strategies and to create the referential constraints:

- The target or the source of the referential constraint must be taken from a list of groups. This list can be made up of all the selected groups or the marked group or all the identifiers (primary or not) or all the schema's groups or a list of groups given by a *Voyager2* function.
- The other end of the referential constraint must match a group of this list. These matching rules are given in the second dialog box described hereafter.
- The Create button is used to create the current referential constraint.
- The Create all button is used to create all the suggested referential constraints.
- The Remove button is used to remove the current referential constraint from the list of suggested referential constraints.
- The Advanced button calls a *Voyager2* procedure that receives all the suggested referential constraints as input parameter. This procedure can be used to create the referential constraints, print a report, query the database, etc.

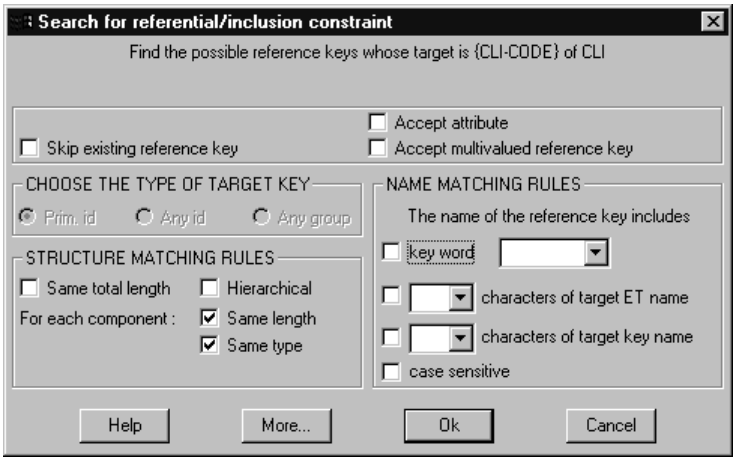


FIGURE 126. Referential constraint assistant dialog (search engine).

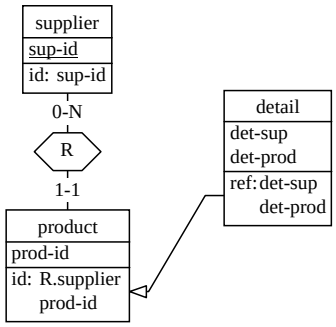


FIGURE 127. Example of a "hierarchical" referential constraint.

The matching rules (figure 123) are the following:

- The type of the target of the foreign key (prim id, any id and any group).
- Structure matching rules: both ends have the same total length or each component has the same length and/or same type. The hierarchical matching rule is used when the identifier may contain a role as in CODASYL databases. For example, in figure 127, a product is identified by the `supplier` of the `product` (the role) and a number (`prod-id`). In `detail`, if we want to create a foreign key that reference a `product`, we need to reference `sup-id` (`det-sup`) and `prod-id` (`det-prod`).
- Name matching rules: the name of the reference key must include a keyword, some (or all) the characters of the target entity type name or some (or all) the characters of the target key name.
- Skip existing reference key.
- Accept attributes: the source of the foreign key is not a group.
- Accept multivalued foreign key.
- More...: the analyst can define his own matching function in *Voyager2*.

When all the strategy parameters are given, the first dialog box displays the list of possible referential constraints. To create one of these referential constraints, the analyst can select it and click on the **Create** button. If he wants to create all of them, he can click on the **Create all** button.

8.3.7. Schema and object integration

DB-MAIN offers two integration tools. The first one integrates two schemas and relies only on the name and type of the objects. It integrates two entity types (or relationship types) if they have the same name and then applies the same rule for their attributes. It also produces an integration report that contains the list of the objects integrated as well as additional information.

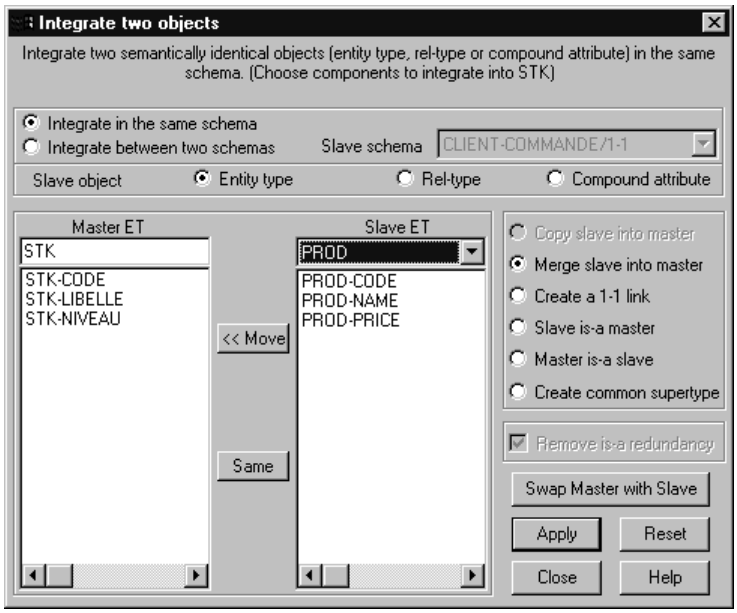


FIGURE 128. The integration assistant.

Schema integration also requires processors that are able to detect semantic correspondences. In the second integration assistant (figure 128), the analyst selects two objects (in the same schema or not) he wants to integrate. He chooses the strategy he wants to use (merge one of the objects in the other, create a common super-type, link them by a one-to-one relationship type, etc.) and for each attribute or role, he decides if it has a counterpart in the other object. This integration assistant can give very precise results but heavily relies on the analyst's knowledge of the application domain.

8.3.8. Schema analysis

The schema analysis assistant is dedicated to the structural analysis of schemas. It uses the concept of submodel, defined as a restriction of the generic specification model. This restriction is expressed by a boolean expression of elementary predicates stating which specification patterns are valid, and which ones are forbidden. An elementary predicate can specify situations such as the following: "entity types must have from 1 to 100 attributes", "relationship types have from 2 to 2 (exactly 2) roles", "entity type names are less than 18 characters long", "names do not include spaces", "there are no compound attributes", "there are no access keys". A submodel appears as a script that can be saved and loaded. Predefined submodels are available: normalized ER, binary ER, relational, CODASYL, etc. Customized predicates can be added via *Voyager2* functions.

The schema analysis assistant offers two functions, namely *check* and *search*. Checking a schema consist in detecting all the constructs which violate the selected submodel while the search function detects all the constructs which comply with the selected submodel.

8.3.9. Transformation toolkit

DB-MAIN proposes a three-level transformation toolset that can be used freely, according to the skill of the user and the complexity of the problem to be solved. These tools are neutral and generic, in that they can be used in any database engineering process. As far as DBRE is concerned, they are mainly used in data structure conceptualization processes. More precisely, the following three levels of transformation are available.

- *Elementary transformations*

  Transformation T is applied to selected object O.

  With these tools, the user keeps full control on the schema transformation since similar situations can be solved by different transformations. E.g, a multivalued attribute can be transformed in many different ways (into an entity type by value or by instance, into a list of single attributes, into a long single attribute, etc.). The current version of DB-MAIN offers a toolset of about 30 elementary transformations.

- *Global transformations*

  Transformation T is applied to all the objects of a schema that satisfy predicate P.

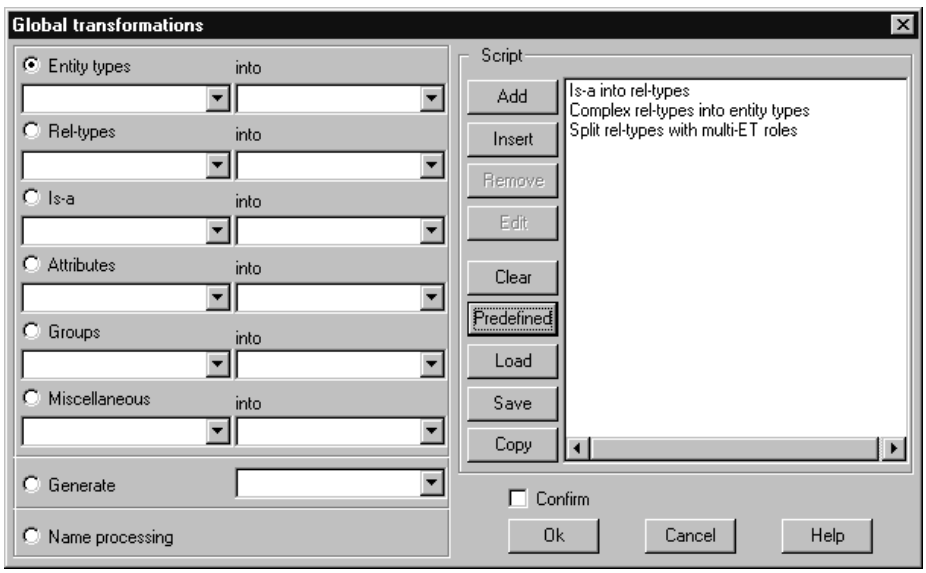


FIGURE 129. The global transformation assistant.

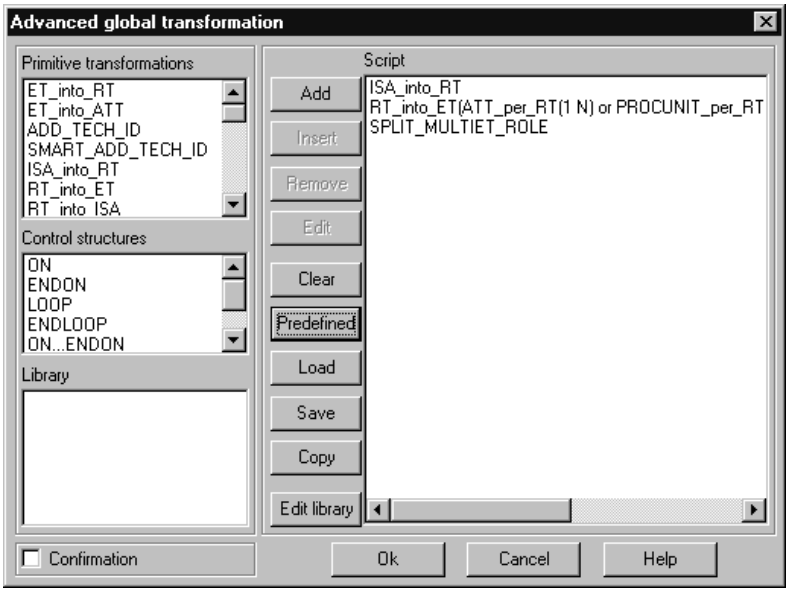


FIGURE 130. The global advanced transformation assistant.

Such a transformation is carried out through a processor that allows the analyst to define T and P independently. DB-MAIN offers two such tools. The first one, the *global transformation* assistant (figure 129), offers a list of predefined predicates with their corresponding transformation (transform all the "relationship entity types" into relationship types, transform all referential constraints into relationship types, etc.). In the second one, the *advanced global transformation* assistant (figure 130), the analyst select a transformation and defines a selection predicate to express on which object the transformation must be applied.

- *Model-driven transformations.*

  All the constructs of a schema that do not comply with a given model are processed through a transformation plan.

  Such an operator is defined by a transformation plan, which is an algorithm comprising global transformations, which is proved (or assumed) to make any schema comply with the model. The DB-MAIN global transformation assistants offer scripting facilities through which the analyst can develop his own transformation plan. Some predefined script are also provided (to transform a conceptual schema into a relation schema, to transform a relational schema into a conceptual schema, etc.).

8.3.10.Graph visualization

The processing schema can be used to represent call graphs, usage graphs, etc. DB-MAIN offers the necessary tools to manipulate (data and processing) schema and to mark and color their elements. The analyst can add dynamic properties, stereotypes and annotations to represent the different kinds of nodes (processing units can represent programs, procedures or modules) and edges.

Creating call graphs by hand can be very long and painful. To automate such graph creation from the program sources code, specific extractors are needed but such extractors can be difficult to write and need to be change for each language. To fulfil this, a *Voyager2* program (`graph_tr.oxo`) has been developed that reads an input file, which describes the schema. It can be used to create any

processing schema that contains processing units, data-objects, call edges, decomposition edges and input/output edges. Such a file can be easily created with scripting language, such as grep/awk or perl. The file format is neutral (see its description in annex A section *A.6*), so it can be used for any graph and it is easily generated from any programs or other JCL scripts.

CHAPTER 9 *Case study*

This chapter presents three small case studies. These case studies are not real programs but we have designed them to illustrate some difficulties that are meted in real projects. The first two recover the complete physical and conceptual schemas of the files used by the same COBOL program. The first one does it manually while the second one does it semi-automatically.

The first example shows the amount of work necessary to reverse engineer a small application. The analyst has to use the same tools several times and the tools he uses do not provide him with the constraints he is looking for. He still has to analyze manually the source code to recover the constraints. An advantage of this approach is that he gains an in-depth knowledge of the application. This knowledge can allow him to retrieve some constraints such as exact cardinality of arrays.

The second example illustrates the fact that the analyst can obtain hints about specific constraints faster. But to interpret the results of the automatic tools, he has to understand how the results were generated. One of the limitations of the automatic approach is that not all the constraints can be recovered this way.

The last case study concerns a COBOL program with embedded SQL. The program offers the same functionalities as the programs of the first two case studies, but it uses a SQL database instead of COBOL files to store the data. This case study illustrates the difficulty to analyze programs with embedded instructions. The analyst has to work in two phases. In the first one, he retrieves the constraints contained in the embedded instructions. During the second phase, he analyzes the link between the embedded code and the host language.

The last part of this chapter is devoted to a brief overview of the real case studies that we have performed in companies. We do not give the code nor the database. We do not present the resolution of the case studies themselves neither for evident confidentiality and space limitation. We present the context of each case study: the language analyzed, the size of the system, the constraints searched for and the results obtained.

9.1. COBOL DBRE, manual process

This section describes how to perform a small COBOL DBRE project manually. The expected results of this project is to produce the complete logical schema and the conceptual schema. The project is not performed entirely manually we use the tools offered by the DB-MAIN CASE tool (DDL extractors, variable dependency graph, program slicing and transformation toolbox).

We have two sources of information, the source code and the data. The source code is a small (300 LOC) COBOL program that manages the customers, the products and the orders of a hypothetical company. The data are stored into indexed COBOL files. We can access the current data to make some tests and we can rely on the fact that the data contain no errors.

9.1.1. Project preparation

In this project, the preparation process is very simple since there is only one source code file (see the complete source code in annex B section B.1) and the files that contain the data.

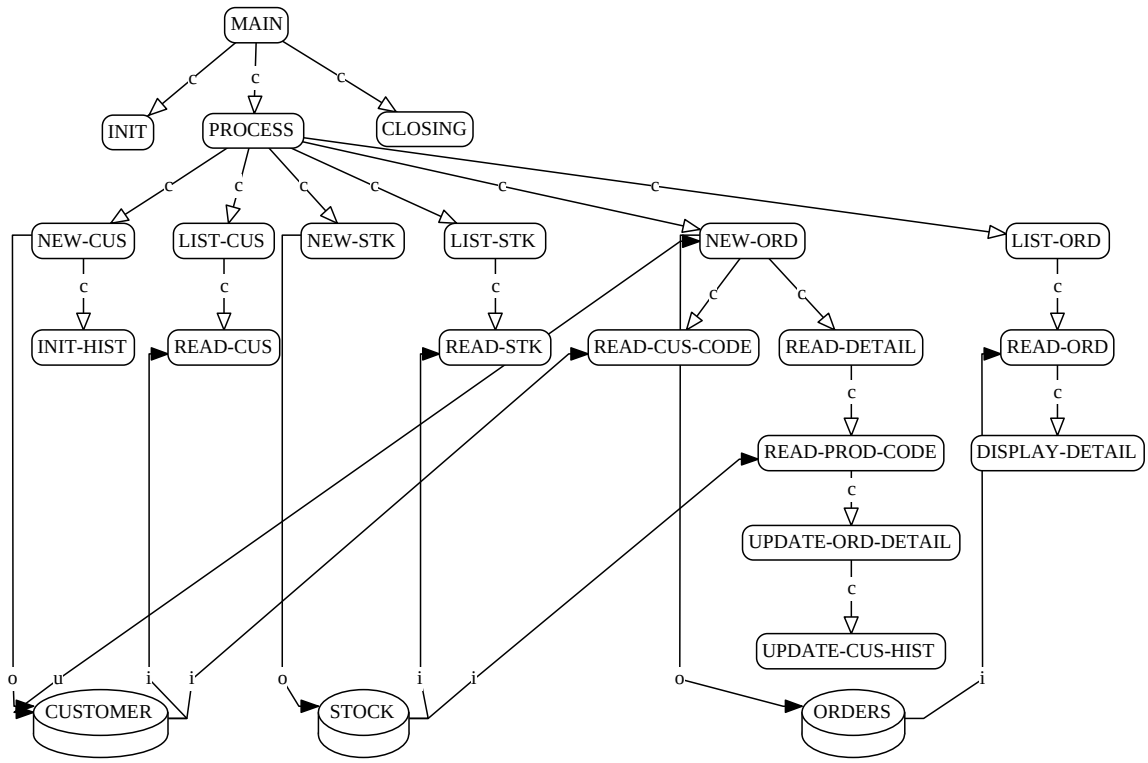


FIGURE 131. General architecture of the program, i.e. the procedure call and data usage graph.

The procedure call graph and data usage graph (figure 131) can be used to get a general overview of the program and to identify the paragraphs that access the data.

9.1.2. Data structure extraction

9.1.2.1. DDL code analysis

The COBOL DDL code is composed of two distinct parts of the program source:

- The `file-control` paragraphs of the `input-output` section of the `environment` division (lines 6-20) declare the files used as well as their organization, their access keys and their identifiers.
- The `FD` paragraphs of the `file` section of the `data` division (lines 24-40) declare the record types (called physical entity types) with their fields (physical attribute) decomposition and the type and the length of the fields.

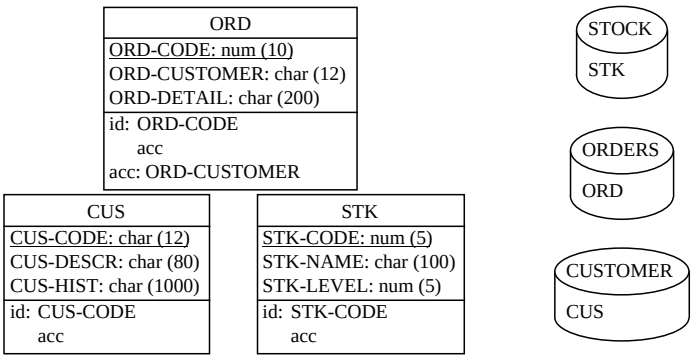


FIGURE 132. The raw physical schema extracted from the COBOL program.

Figure 132 represents the raw physical schema obtained by the DB-MAIN COBOL extractor (**File - Extract - COBOL**).

9.1.2.2. Physical integration

There is only one raw physical schema in this project, so the physical integration is needed.

9.1.2.3. Schema refinement

During schema refinement, the physical schema is analyzed to discover hypotheses about the fine-grained structure of entity types and attributes, finding referential constraints, finding sets behind arrays, finding exact cardinalities of attributes and finding identifiers of multivalued attributes. Each hypothesis will be validated through program code and data analysis. Validated hypothesis will be added to the schema.

A. Finding the fine-grained structure of entity types and attributes

Some attributes have an unusual length, which suggests that it could be possible to find a fine-grained structure for them.

Hypothesis discovery. In this example, We state that the attributes that have a length greater than 50 are good candidates to be refined. This rule concerns *CUS-DESC*, *CUS-HIST*, *ORD-DETAIL* and *STK-NAME*.

Of course the limit of 50 is completely arbitrary. Attributes longer than 50 may be atomic, such as a name or a book title. On the other hand, attributes smaller than 50 can be compound such as a date that is stored in a 8 character variable, that can be decomposed in day, month and year.

Hypothesis validation. To validate a decomposition hypothesis, we will use the variable dependency graph computed for assignment instructions. The dependency graph (**Assist - Text analysis - Dependency**) is computed for the pattern `"move" - @var_1 - "to" - @var_2`. As explained in chapter 8, section 8.3.4, DB-MAIN displays the variable dependency graph in context, that is, in the source code.

We select one by one the "long" attributes and verify to which other variables each of them is connected. If a "long" attribute is connected, directly or indirectly, to a variable that has a fine-grained structure, then the hypothesis is validated and the structure of the variable is assigned to the attribute. The analysis of the variable dependency graph shows us that:

- `CUS-DESC` is connected to `DESCRIPTION` and `DESCRIPTION` has the following decomposition (line 43 - 47)
```
01 DESCRIPTION.
   02 NAME PIC X(20).
   02 ADDR PIC X(40).
   02 FUNCT PIC X(10).
   02 REC-DATE PIC X(10).
```
- `CUS-HIST` is connected to `LIST-PURCHASE` and `LIST-PURCHASE` has the following decomposition (line 49 - 52)
```
01 LIST-PURCHASE.
   02 PURCH OCCURS 100 TIMES INDEXED BY IND.
      03 REF-PURCH-STK PIC 9(5).
      03 TOT PIC 9(5).
```
- `ORD-DETAIL` is connected to `LIST-DETAIL` and `LIST-DETAIL` has the following decomposition (line 54 - 57)
```
01 LIST-DETAIL.
   02 DETAILS OCCURS 20 TIMES INDEXED BY IND-DET.
      03 REF-DET-STK PIC 9(5).
      03 ORD-QTY PIC 9(5).
```
- `STK-NAME` is not present in the graph, because this attribute does not appear in any assignment, so we can conclude that there is no decomposition for `STK-NAME`.

CUS
CUS-CODE: char (12)
CUS-DESCR: compound (80)
NAME: char (20)
ADDR: char (40)
FUNCT: char (10)
REC-DATE: char (10)
CUS-HIST: compound (1000)
PURCH[100-100] array: compound (10)
REF-PURCH-STK: num (5)
TOT: num (5)
id: CUS-CODE
acc

ORD
ORD-CODE: num (10)
ORD-CUSTOMER: char (12)
ORD-DETAIL: compound (200)
DETAILS[20-20] array: compound (10)
REF-DET-STK: num (5)
ORD-QTY: num (5)
id: ORD-CODE
acc
acc: ORD-CUSTOMER

STK
STK-CODE: num (5)
STK-NAME: char (100)
STK-LEVEL: num (5)
id: STK-CODE
acc

FIGURE 133. The schema with fine-grained structure.

Schema enhancement. The schema can be enhanced with the new decompositions of *CUS-DESC*, *CUS-HIST* and *ORD-DETAIL* (figure 133).

B. Finding referential constraints

Though there are no referential constraints or relationship types between the different entity types, we can guess that such links should exist between *CUSTOMER*, *ORDER* and *STOCK*.

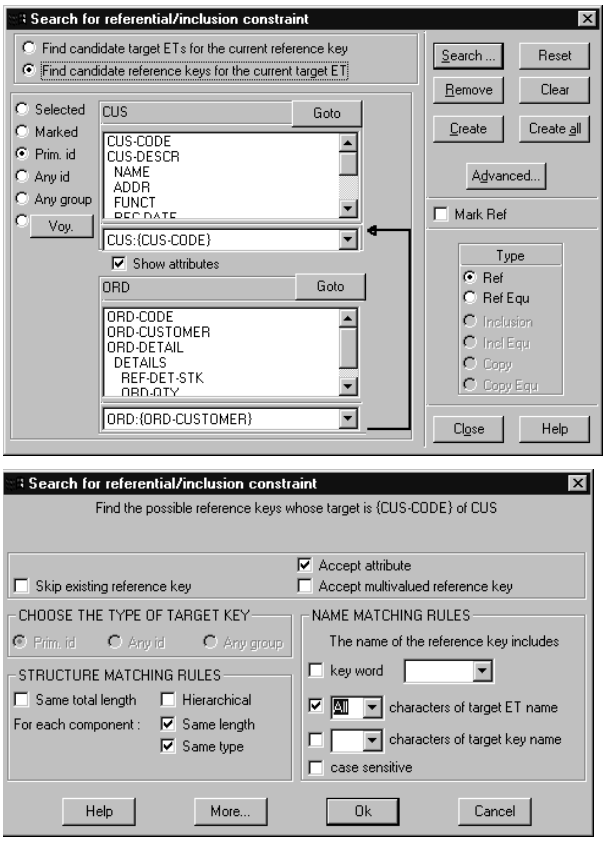


FIGURE 134. The configuration of the referential constraint assistant to discover the potential referential constraints.

Hypothesis discovery. We make the assumption that the program was well designed and that some naming conventions have been used. To discover potential referential constraints, we analyze the schema to find potential referential constraints that have an identifier as target, both sides have the same type and same length and the name of the referential attribute contains the name of the target entity type.

The referential constraint assistant (**Assist - Referential key**) can be used to discover such potential referential constraints. The figure 134 shows the configuration of the assistant to perform this task. The assistant suggests the following potential referential constraints, specified by their source and target attributes:

- `ORD.ORD-CUSTOMER` → `CUS.CUS-CODE`
- `CUS.CUS-HIST.PURCH.REF-PURCH-STK` → `STK.STK-CODE`
- `ORD.ORD-DETAIL.DETAILS.REF-DET-STK` → `STK.STK-CODE`
- `STK.STK-LEVEL` → `STK.STK-CODE`

Hypothesis validation. We then use program slicing. To validate a referential constraint, the analyst must verify that before each (`RE`)`WRITE` instruction of the source entity type, the referential constraint is verified. For this, we compute the program slice with respect to the write instruction of the entity type and the referential attribute origin. If the slice contains an instruction that reads the target entity type and there is a validation of the value of the referential attribute (it is an existing value of the target identifier), then the referential constraint is assumed to be verified.

```
183   MOVE 1 TO END-FILE.                203 READ-CUS-CODE.
184   PERFORM READ-CUS-CODE              204   DISPLAY "CUSTOMER NUMBER "
        UNTIL END-FILE=0.                205     WITH NO ADVANCING.
187   MOVE CUS-CODE TO ORD-CUSTOMER.     206   ACCEPT CUS-CODE.
196   WRITE ORD                          207   MOVE 0 TO END-FILE.
                                         208   READ CUSTOMER INVALID KEY
                                         209     DISPLAY "NO SUCH CUSTOMER"
                                         210     MOVE 1 TO END-FILE
                                         211   END-READ.
```

FIGURE 135. Program slice with respect to `write ORD` (line 196) and `ORD-CUSTOMER`.

ORD.ORD-CUSTOMER → CUS.CUS-CODE

To validate the first referential constraint (*ORD.ORD-CUSTOMER → CUS.CUS-CODE*), we search all the instructions that modify *ORD* ((`RE`)`WRITE ORD`). In this program there is only one such instruction at line 196. We compute the program slice with respect to the `write ORD` instruction (line 196) and `ORD-CUSTOMER`. The program slice is displayed in figure 135; we only display (as in the remainder of the chapter) the line of the slice and the line with respect to which the slice is computed is in bold. The slice shows that the procedure `READ-CUS-CODE` is executed until `END-FILE` is equal to `0` (line 184). In `READ-CUS-CODE`, the user is asked for a `CUS-CODE` value (line 206) and the file `CUSTOMER` is read to check if this value of `CUS-CODE` exist otherwise `END-FILE` is set to 1 (line 208-211). So after the execution of `READ-CUS-CODE`, `CUS-CODE` contains a value that exists in the file `CUSTOMER`. `CUS-CODE` is copied to `ORD-CUSTOMER` (line 187) and the record is written into the file. This proves the existence of the referential constraint.

```
190   SET IND-DET TO 1.                    233 UPDATE-ORD-DETAIL.
191   MOVE 1 TO END-FILE.                  234   MOVE 1 TO NEXT-DET.
192   PERFORM READ-DETAIL                  235   DISPLAY "QUANTITY ORDERED "
193    UNTIL END-FILE = 0 OR IND-DET=21.   236     WITH NO ADVANCING
194   MOVE LIST-DETAIL TO ORD-DETAIL.      237   ACCEPT ORD-QTY(IND-DET).
196   WRITE ORD                            238   PERFORM UNTIL
213 READ-DETAIL.                           239    (NEXT-DET < IND-DET AND
214   DISPLAY "PRODUCT CODE (0=END):".     240     REF-DET-STK(NEXT-DET)=PROD-CODE)
215   ACCEPT PROD-CODE.                    241    OR IND-DET = NEXT-DET
216   IF PROD-CODE = 0                     242     ADD 1 TO NEXT-DET
217     MOVE 0 TO REF-DET-STK(IND-DET)     243   END-PERFORM.
219     MOVE 0 TO END-FILE                 244   IF IND-DET = NEXT-DET
220   ELSE                                 245     MOVE PROD-CODE
221     PERFORM READ-PROD-CODE.            246       TO REF-DET-STK(IND-DET)
223 READ-PROD-CODE.                        248     SET IND-DET UP BY 1
224   MOVE 1 TO EXIST-PROD.                249   ELSE
225   MOVE PROD-CODE TO STK-CODE.          250     DISPLAY "ERROR: ALREADY ORDERED".
226   READ STOCK INVALID KEY               271 LIST-ORD.
227     MOVE 0 TO EXIST-PROD.              273   CLOSE ORDERS.
228   IF EXIST-PROD = 0                    275   MOVE 1 TO END-FILE.
229     DISPLAY "NO SUCH PRODUCT"          276   PERFORM READ-ORD UNTIL END-FILE=0.
230   ELSE
231      PERFORM UPDATE-ORD-DETAIL.
```

FIGURE 136. Program slice with respect to `write ORD` (line 196) and `ORD-DETAIL`.

ORD.ORD-DETAIL.DETAILS.REF-DET-STK → STK.STK-CODE

To validate the foreign key (*ORD.ORD-DETAIL.DETAILS.REF-DET-STK → STK.STK-CODE*), once again we need to know the instruction that modifies *ORD*, as for the previous referential constraint. We compute the program slice with respect to `write ORD` (line 196) and `ORD-DETAIL` (figure 136). We observe that the `READ-DETAIL` procedure is performed until `END-FILE` equal `0` or `IND-DET` equal `21` (maximum number of elements of `DETAILS` array plus one). In `READ-DETAIL` a product code (`PROD-CODE`) is asked (line 215). If it is `0` then `END-FILE` is set to `0` and the message of line 214 says that "`0 = end`". This mean that the user can stop to specify products when he wants, so the cardinality of the array `DETAILS` actually is [0-20]. If `PROD-CODE` is different of `0`, `READ-PROD-CODE` is performed. `READ-PROD-CODE` checks if `PROD-CODE` is an existing value of `STK-CODE`. If it does not exist in the file `STOCK`, an error message is displayed ("`no such product`", line 229) and a new product number is asked; else `UPDATE-ORD-DETAIL` is performed.

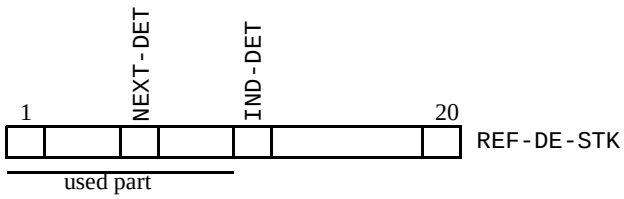


FIGURE 137. The situation at any moment in the loop [238-243].

In `UPDATE-ORD-DETAIL`, the loop [238-243] goes through the `DETAILS` array. `IND-DET` is the index of the first unused cell of `DETAILS` (all the elements before `IND-DET` have a value different of `0`) and `NEXT-DET` is the number of the current element (see figure 137). The loop has two ending conditions:

1. `NEXT-DET < IND-DET AND REF-DET-STK(NEXT-DET) = PROD-CODE`

REF-DET-STK(NEXT-DET) is an element of the already filled part of DETAILS and it is equal to PROD-CODE (the current product), i.e., the customer tries to order a second time the same product in the order.

2. IND-DET = NEXT-DET

It has reached the first non used element of DETAILS.

```
108 NEW-CUS.                                294 INIT-HIST.
126   PERFORM INIT-HIST.                    295   SET IND TO 1.
127   WRITE CUS.                            296   PERFORM UNTIL IND = 100
                                            297     MOVE 0 TO REF-PURCH-STK(IND)
                                            298     MOVE 0 TO TOT(IND)
                                            299     SET IND UP BY 1
                                            300   END-PERFORM.
                                            301   MOVE LIST-PURCHASE TO CUS-HIST.
```

FIGURE 138. Program slice with respect to `write CUS` (line 127) and CUS-HIST.

```
183   MOVE 1 TO END-FILE.                   228   IF EXIST-PROD = 0
184   PERFORM READ-CUS-CODE                 229     DISPLAY "NO SUCH PRODUCT"
      UNTIL END-FILE = 0.                   230   ELSE
188   MOVE CUS-HIST TO LIST-PURCHASE.       231     PERFORM UPDATE-ORD-DETAIL.
190   SET IND-DET TO 1.                     233 UPDATE-ORD-DETAIL.
191   MOVE 1 TO END-FILE.                   234   MOVE 1 TO NEXT-DET.
192   PERFORM READ-DETAIL                   238   PERFORM UNTIL
199   MOVE LIST-PURCHASE                    239     (NEXT-DET < IND-DET AND
200     TO CUS-HIST.                        240     REF-DET-STK(NEXT-DET)=PROD-CODE)
201   REWRITE CUS                           241     OR IND-DET = NEXT-DET
203 READ-CUS-CODE.                          242     ADD 1 TO NEXT-DET
204   DISPLAY "CUSTOMER NUMBER "            243   END-PERFORM.
205     WITH NO ADVANCING.                  244   IF IND-DET = NEXT-DET
206   ACCEPT CUS-CODE.                      247     PERFORM UPDATE-CUS-HIST
207   MOVE 0 TO END-FILE.                   248     SET IND-DET UP BY 1.
208   READ CUSTOMER INVALID KEY             252 UPDATE-CUS-HIST.
209     DISPLAY "NO SUCH CUSTOMER"          253   SET IND TO 1.
210     MOVE 1 TO END-FILE                  254   PERFORM UNTIL
211   END-READ.                             255     REF-PURCH-STK(IND) = PROD-CODE
213 READ-DETAIL.                            256     OR REF-PURCH-STK(IND) = 0
214   DISPLAY "PRODUCT CODE (0=END):".      257     OR IND = 101
215   ACCEPT PROD-CODE.                     258       SET IND UP BY 1
216   IF PROD-CODE = 0                      259   END-PERFORM.
217     MOVE 0                              263   IF REF-PURCH-STK(IND)
218       TO REF-DET-STK(IND-DET)           264       = PROD-CODE
219     MOVE 0 TO END-FILE                  265     ADD ORD-QTY(IND-DET) TO TOT(IND)
220   ELSE                                  266   ELSE
221     PERFORM READ-PROD-CODE.             267     MOVE PROD-CODE
223 READ-PROD-CODE.                         268       TO REF-PURCH-STK(IND)
224   MOVE 1 TO EXIST-PROD.                 269     MOVE ORD-QTY(IND-DET)TO TOT(IND).
225   MOVE PROD-CODE TO STK-CODE.           289   MOVE ORD-DETAIL TO LIST-DETAIL
226   READ STOCK INVALID KEY
227     MOVE 0 TO EXIST-PROD.
```

FIGURE 139. Program slice with respect to `rewrite CUS` (line 201) and CUS-HIST.

If the second condition is satisfied (line 244), then PROD-CODE is assigned to REF-DET-STK(NEXT-DET) (line 245-246) and IND-DET is incremented by one (line 248), otherwise an error message is displayed (line 250). This means that the referential constraint is validated. *REF-DET-*

STK is a local identifier of *DETAILS*, because the user cannot order twice the same product in the same order. *DETAILS* is an array that is managed like a set, because there is no gap in the array (the first attribute is used, then the second, then the third, etc.) and the user can order the product in the order he wants (there is no sequencing criterion).

CUS.CUS-HIST.PURCH.REF-PURCH-STK → STK.STK-CODE

To validate the foreign key (*CUS.CUS-HIST.PURCH.REF-PURCH-STK → STK.STK-CODE*), we have to compute and to analyze two program slices, because there is a `write CUS` at line 127 and a `rewrite CUS` instruction at line 201. We have to analyze both slices to be sure that they validate the same constraints.

Figure 138 shows the program slice with respect to `write CUS` (line 127) and `CUS-HIST`. The procedure `INIT-HIST` is called once and all the element of `PURCH` are set to `0`. `INIT-HIST` initializes the `PURCH` array and `0` is used to represent the null value. So the cardinality of the array is not [100-100] but [0-100]. The referential constraint is trivially verified by the code fragment since the array is empty!

Then, we compute the second program slice with respect to `write CUS` (line 201) and `CUS-HIST` (figure 139). As in the slice of the first referential constraint, after the execution of `READ-CUS-CODE`, `CUS-CODE` contains a validated value. `CUS-HIST` is copied into `LIST-PURCH` (line 188). The analysis of the paragraphs `READ-DETAIL`, `READ-PROD-CODE` and `UPDATE-ORD-DETAIL` is the same as that performed for the second referential constraint. So, when `UPDATE-CUS-HIST` is performed (line 247) `PROD-CODE` contains a validated value of *STK-CODE* and `ORD-QTY(IND-DET)` contains the ordered quantity of the product `PROD-CODE`.

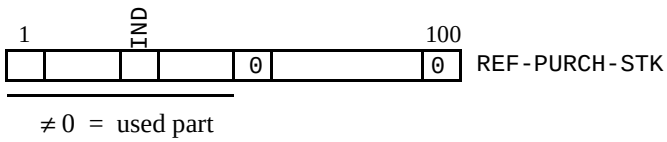


FIGURE 140. The situation at any moment in the loop [254-259].

The loop in `UPDATE-CUS-HIST` (line 254-259) goes through the `PURCH` array. The unused cells of `PURCH` have their `REF-PURCH-STK` component set to `0` and `IND` is the number of the current element (see figure 140). The loop has three ending conditions:

1. `REF-PURCH-STK(IND) = PROD-CODE`

   The loop has found the current product in the customer history (product already ordered).
2. `REF-PURCH-STK(IND) = 0`

   It has reached the first `REF-PURCH-STK` equal to 0, i.e.; the product is not present in the customer history (product not already ordered).
3. `IND = 101`

   It has reached the end of the array.

`ORD-QTY(IND-DET)` (the ordered quantity) is added to `TOT(IND)` (line 265) if `PROD-CODE` is already present in the `REF-PURCH-STK` (at the `IND` element), otherwise the first free element of the array (`IND`) is filled with the `PROD-CODE` (line 267) and ordered quantity (line 269). This validates the referential constraint (all the `REF-PURCH-STK` not equal to `0` are valid values of `STK-CODE`) The exact cardinality of the array is [0-100], because there can be unused elements filled with `0`.

REF-PURCH-STK is a local identifier of LIST-PURCH, because when the loop has found a matching element, it stops searching, as if no other similar element could be found. With the same reasoning as for DETAILS, we show that PURCH actually is a set.

```
145 NEW-STK.
154   DISPLAY "LEVEL "
       WITH NO ADVANCING.
155   ACCEPT STK-LEVEL.
157   WRITE STK
```

FIGURE 141. Program slice with respect to `write STK` (line 157) and STK-LEVEL.

STK.STK-LEVEL → STK.STK-CODE

The last referential constraint to check is (*STK.STK-LEVEL, STK.STK-CODE*). Without any program analysis, it can be seen that STK-LEVEL cannot be a foreign key. The matching rule is satisfied because COBOL programmers often prefix the field name by the name of the record type to get unique names. But to be sure, we can compute the program slice with respect to `write STK` (line 157) and STK-LEVEL (figure 141). In the slice, we can see that STK-LEVEL value is given by the user (line 154) and there no validation is carried out

Logical schema validation. The referential constraints and the local identifier of the multivalued attributes can also be validated through the data analysis. To analyze the data, we write a COBOL program that queries the file contents and produces a report.

Schema enhancement. Through the validation of the hypothesis about the referential constraints, we have discovered three referential constraints and several other properties about the multivalued attributes:

- The three referential constraints are (*ORD.ORD-CUSTOMER*, *CUS.CUS-CODE*), (*CUS.CUS-HIST.PURCH.REF-PURCH-STK*, *STK.STK-CODE*) and (*ORD.ORD-DETAIL.DETAILS.REF-DET-STK*, *STK.STK-CODE*).
- The exact cardinality of *DETAILS* is [0-20].
- *REF-DET-STK* is the local identifier of *DETAILS*.
- *DETAILS* is a set and not an array.
- The exact cardinality of *PURCH* is [0-100].
- *REF-PURCH-STK* is the local identifier of *PURCH*.
- *PURCH* is a set and not an array.

We were not looking for multivalued attributes properties, but we discovered them during the validation of the referential constraints. This is called the opportunistic approach.

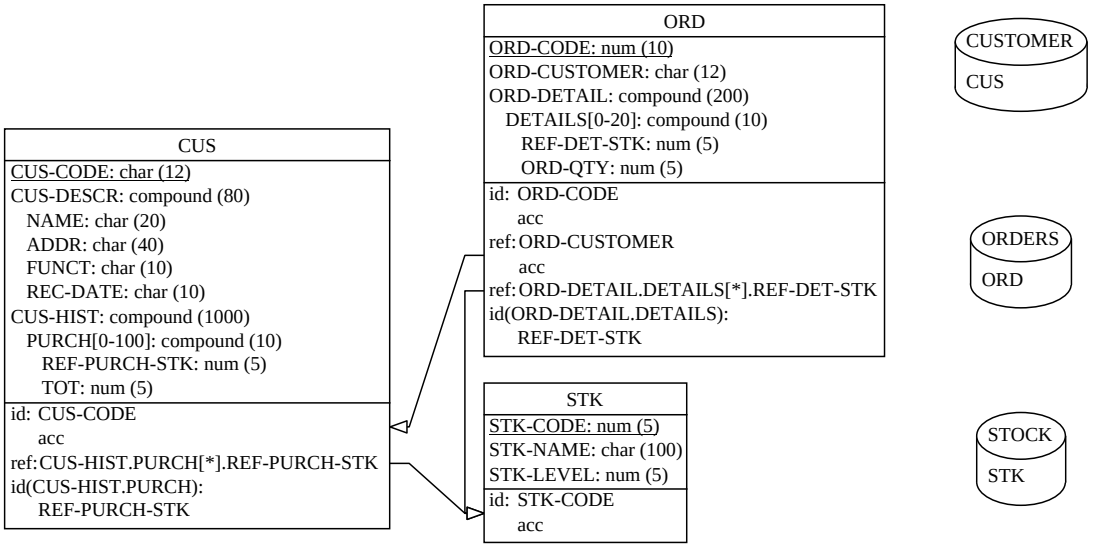


FIGURE 142. The complete physical schema.

All the discovered constraints are added to the physical schema to obtain the complete physical schema (figure 142).

9.1.2.4. Schema cleaning

Considering that the logical schema includes all the constraints that must be known by the programmer, the logical schema of a COBOL set of files includes access keys and entity collections (files).

The physical schema of figure 142 is also the complete logical schema.

9.1.3. Data structure conceptualization

9.1.3.1. Preparation

This phase prepares the schema such that it contains only structures and constraints that are necessary to understand the semantics of the schema.

A. Name processing

The name of the objects are the names given by the programmers (as recovered during data structure extraction), who have used some naming rules. Now the names can be changed to give more information on the named objects:

- *Remove common prefixes*

  A common naming conversion in COBOL consists in prefixing each attribute name by the name (or a short name) of the entity type. This is useful in large programs to ensure the uniqueness of the attribute names. Those prefixes do not give any information, so that they can be removed (**Transform - Change prefix**).

- *Meaningful name*

  The names of the collections are more meaningful than the corresponding entity types names, so that the entity types name can be replaced by the collections name.

B. Abnormal structures transformation

During the data structure extraction, some compound attributes with only one component were created (*CUS-HIST* and *ORD-DETAIL*). It is suggested to disaggregate them (**Transform - Attribute - Disaggregation**) to remove unnecessary levels of decomposition.

C. Discard the physical constructs

The access keys and collections are not useful any more, and can be suppressed.

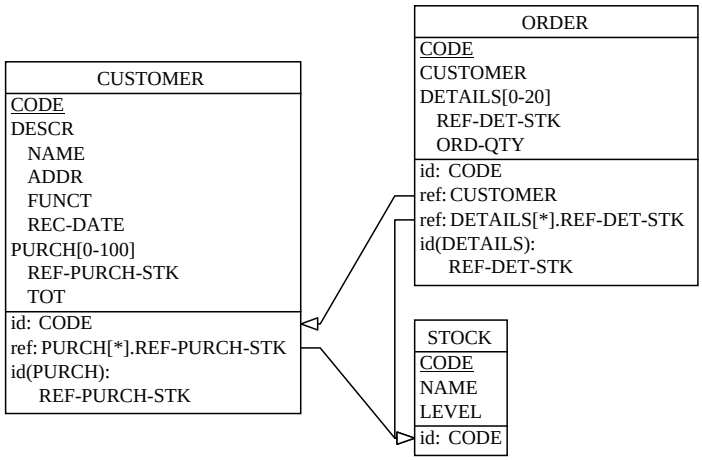


FIGURE 143. The prepared logical schema.

The result of these transformations gives the prepared logical schema, as displayed in figure 143.

9.1.3.2. Basic conceptualization

The de-optimization and untranslation processes are generally interleaved, so that it can be artificial to try to dissociate them.

A. Complex multivalued attributes

The two complex (decomposable, multivalued, with local identifier and foreign key) attributes *DETAILS* and *PURCH* are typical implementations of dependent entity types. This structure is used to decrease the number of files used and the number of disk access. This is a common optimization that can be undone by the transformation of *DETAILS* and *PURCH* into entity types (**Transform - Attribute - Entity type**).

B. Foreign keys

Since foreign keys implement relationship types, they are transformed into relationship types (**Transform - Group - Rel-type**).

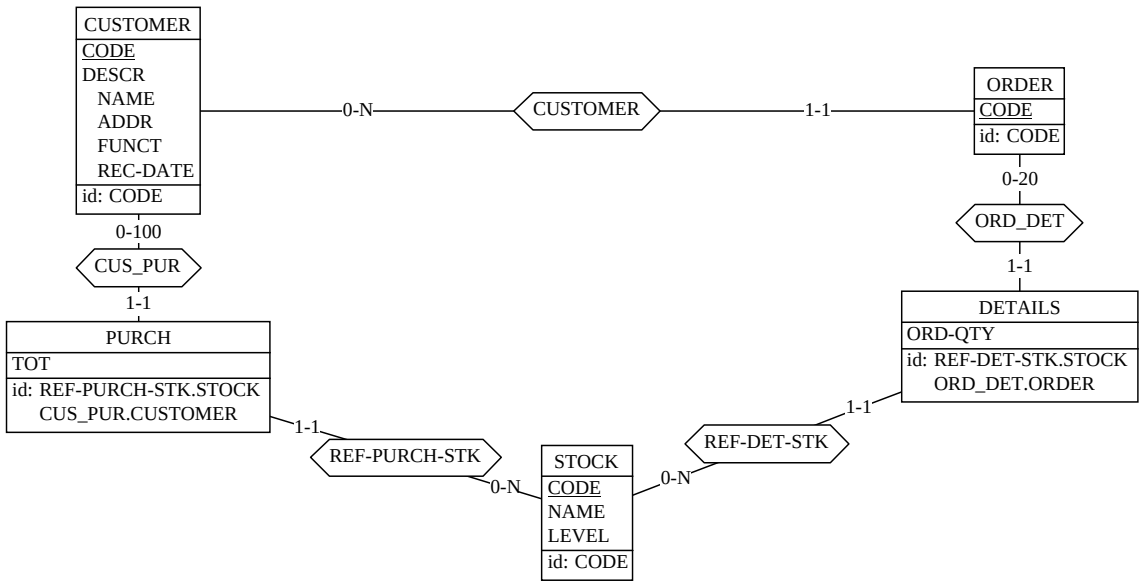


FIGURE 144. The raw conceptual schema.

These transformations produce the raw conceptual schema (figure 144).

9.1.3.3. Normalization

Through the normalization, the analyst tries to give the schema such qualities as readability, concision, minimality, expressiveness, typical to good conceptual schemas.

A. Relationship entity types

The entity types `PURCH` and `DETAILS` can be perceived as playing the role of relationship types between `CUSTOMER` and `STOCK` and between `ORDER` and `STOCK`, respectively. Those entity types can be transformed into relationship types (**Transform - Entity type - Rel-type**).

B. Attribute disaggregation

The attribute `DESC` is composed of attributes of different domain concepts, we can decide to disaggregate them (**Transform - Attribute - Disaggregation**).

C. Name processing

Some names can be changed to be more meaningful. For example, the attribute *TOT* can be changed to *TOTAL*; *ADDR* can be changed to *ADDRESS*; *FUNCT* can be changed to *FUNCTION*;

ORD-QTY can be changed to *QUANTITY*; *PURCH* can be changed to *purchase*; *CUSTOMER* relationship type can be changed to *pass*.

The attribute *DETAILS* is a plural and the habit is to use the singular. It is replaced by *DETAIL*.

An usual naming rule is to use lowercase for relationship type names, uppercase for the entity type names and capitalized for the attribute names.

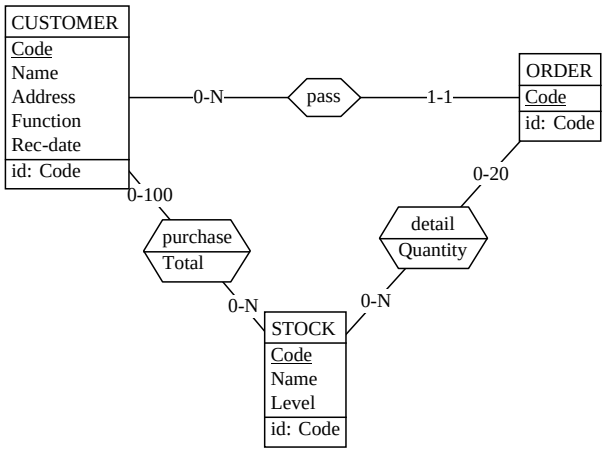


FIGURE 145. The normalized conceptual schema.

The final conceptual schema is shown in figure 145.

9.2. COBOL DBRE, (semi-)automatic process

This section shows how to recover (semi)-automatically the conceptual schema of the first case study.

9.2.1. Data structure extraction

The DDL code analysis is the same as in the previous case study (see 9.1.2.1). The raw physical schema is that of figure 132. The main difference is that we will increase the level of automation of the schema refinement process.

9.2.1.1. Schema refinement

A. Finding the fine-grained structure of entity types and attributes

Finding the fine-grained structure of attributes can be partially automated.

In the previous case study (9.1.2.3) we had to select one by one the attributes to see if they are connected, in the variable dependency graph, to a variable with a more precise decomposition. DB-MAIN offers the possibilities to store the variable dependency graph as a text file. A *Voyager2*

program (`depend.oxo`) has been written to read this file and to produce a graphical representation of the graph.

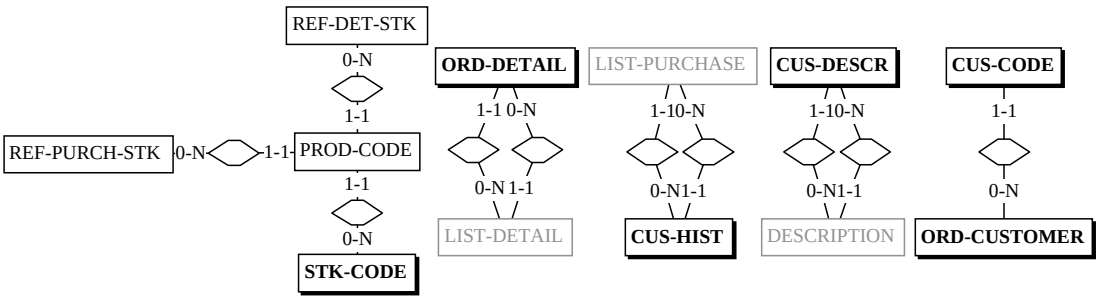


FIGURE 146. The variable dependency graph.

Since DB-MAIN has no specific view to represent a variable dependency graph, we represent it as an entity-relationship schema, where the variables are entity type and the relations between variables are relationship types (see annex A, section A.4.3 for a complete description of `depend.oxo`). The variable dependency graph obtained is displayed as in figure 146. In this schema, the variables that represent an attribute of the raw physical schema are in bold. The compound variables are in grey.

The analysis of this graph shows that a candidate decomposition exists for the attributes *ORD-DETAIL*, *CUS-HIST* and *CUS-DESCR*. The analyst has to find (manually) the correct decomposition for those attributes and to refine them.

B. Finding referential constraints

To find referential constraints, a program slice is computed for each `write` (or `rewrite`) instruction and it is searched for a `read` instruction. When a `read` instruction is found in a slice computed with respect to a `write`, the dataflow is analyzed to know which attributes are in relation.

To automate this as much as possible, the command line version of the program slicing is used (see section 8.3.5 and annex A section A.5.3 for a complete description of this tool). One of the possible usage of the command line program slicing is the following:

```
slicing -v -s write -c read -a var -o order.dep order.cob
```

The options of this command can be interpreted as follow. `-v` computes the variable follow-up program slicing. `-s write` computes the program slicing with respect to each `write` (or `rewrite`) instruction and its record. `-c read` checks if there is a `read` (or `start`) instruction in the slice. If the slice does not contains a `read` instruction the slice is not memorized. `-a var` option is used to display (into the output file) the variables of the `read` and `write` instruction between which there is a dataflow. `-o order.dep` specifies that the output must be save into the `order.dep` file.

ataflow	READ				(RE)WRITE				
	Line	Entity type	Start-stop	Att	Line	Entity type	Start-stop	Att	Lines
D	208	CUSTOMER	(1-12)	cus-code	196	ORDERS	(11-22)	ord-customer	208,187,196
D+C	226	STOCK	(1-5)	stk-code	196	ORDERS	(23-27)	ord-detail	226,227,228,230,231,233, 245, 246,194,196
D+C	226	STOCK	(1-5)	stk-code	201	CUSTOMER	(93-97)	cus-hist	226,227,228,231,233,244, 247,252,263,264,267,268, 199, 200,201
D	208	CUSTOMER	(93-97)	cus-hist	201	CUSTOMER	(93-97)	cus-hist	208,188,263,264,266,267, 268, 199,200,201

FIGURE 147. Result of the command line program slicing on `order.cob`

A summary of the `order.dep` file is given in figure 147. The first column says if only dataflow has been used to find the relation (value = `D`) or if control flow has also been used (value = D+C). The second (resp. third) column gives information about the `READ` (resp. `(RE)WRITE`) instruction. The first sub-column is the line number of the `READ` (resp. `(RE)WRITE`) instruction. The third sub-column is the begin and end position of the read (modified) attribute relative to the entity type of the second column. The fourth sub-column is the name of the attribute. The last column gives the line numbers of the instructions of the slice used to detect the relation between the two entity types.

A close look at the table shows that in the first three lines, the attribute read is always an identifier. These three lines mean that the identifier of an entity type is read and its value influences the value of the attribute of another entity type. This is a strong hint that a referential constraint exists. To be sure, we need to analyze the program slice used to find those relations. The lines of the program slice are shown in the last column. They are a sub-set of the lines used in the previous case study to find the referential constraints (figure 138, figure 139 and figure 141).

It can be noticed that the start-stop of the `(RE)WRITE` of the second and third lines do not span the entire field (`ord-detail` and `cus-hist`). This is a hint that these fields have a meaningful decomposition. When the dataflow goes through the elements of an array (since program slicing computes a static slice) program slicing does not know which index is used, so it maps all the accesses to an array to its first element. For example, if we have the following variable declaration

```
01 my-array occurs 3.
  02 A1 pic X(3).
  02 A2 pic X(5).
```

and if there is the following assignation

```
move X to A1(I).
```

Then the program slicing tool assumes the following position of the variable `my-array` are modified:

```
my-array(1-3)
```

To know exactly which are the referential attributes, the line used to detect the dataflow has to be analyzed. The program slice does not only use dataflow to find these referential constraints, it also use some control flow edges in the SDG.

```
177  NEW-ORD.                                223  READ-PROD-CODE.
192   PERFORM READ-DETAIL                    226    READ STOCK INVALID KEY
193     UNTIL END-FILE = 0                   225    MOVE PROD-CODE TO STK-CODE.
              OR IND-DET = 21.                227      MOVE 0 TO EXIST-PROD.
194    MOVE LIST-DETAIL TO ORD-DETAIL.        228    IF EXIST-PROD = 0
196    WRITE ORD                              230    ELSE
213  READ-DETAIL.                            231       PERFORM UPDATE-ORD-DETAIL.
215    ACCEPT PROD-CODE.                     233  UPDATE-ORD-DETAIL.
216    IF PROD-CODE = 0                      245      MOVE PROD-CODE
220    ELSE                                  246      TO REF-DET-STK(IND-DET)
221      PERFORM READ-PROD-CODE.
```

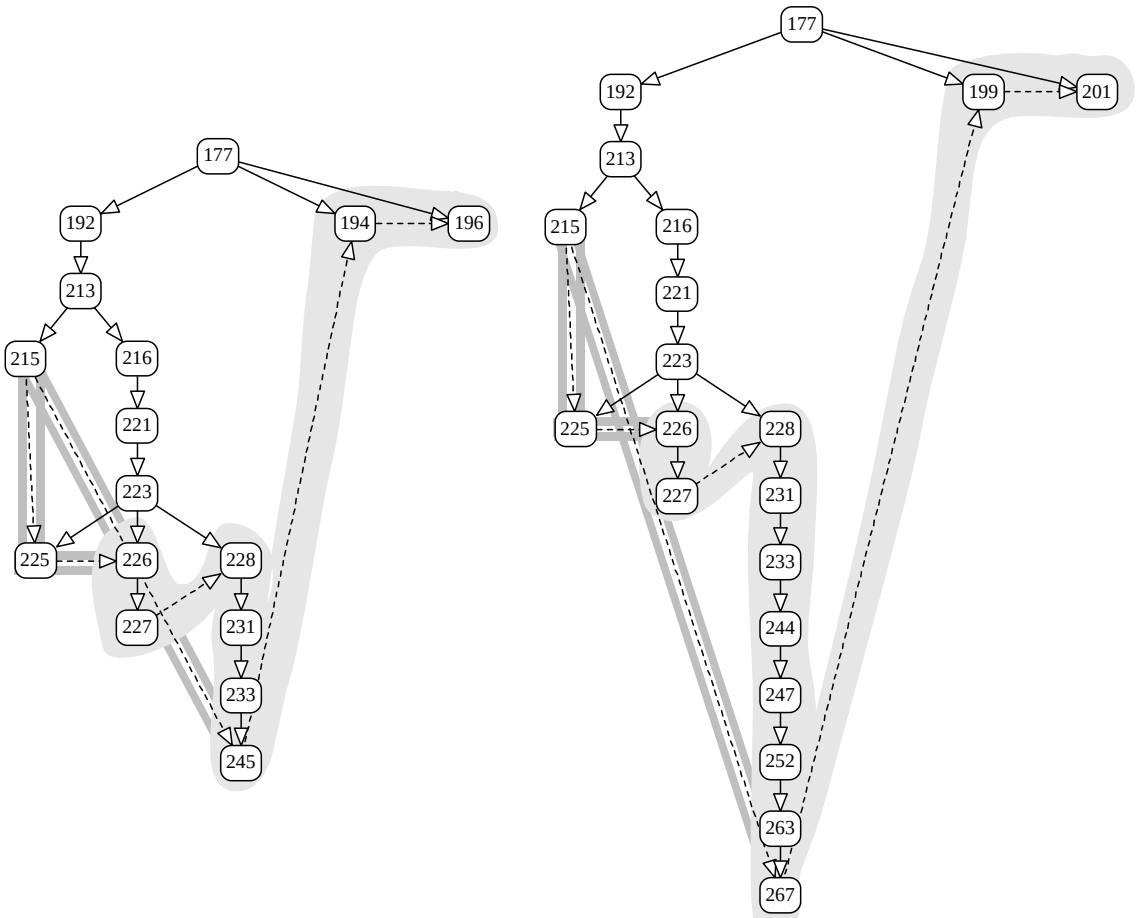
FIGURE 148. The code fragment used to detect the referential constraint between ORD and STK.

```
177  NEW-ORD.                                223  READ-PROD-CODE.
192   PERFORM READ-DETAIL                    225    MOVE PROD-CODE TO STK-CODE.
199    MOVE LIST-PURCHASE                    226    READ STOCK INVALID KEY
200      TO CUS-HIST.                        227      MOVE 0 TO EXIST-PROD.
201    REWRITE CUS                           228    IF EXIST-PROD = 0
213  READ-DETAIL.                            231       PERFORM UPDATE-ORD-DETAIL.
215    ACCEPT PROD-CODE.                     233  UPDATE-ORD-DETAIL.
216    IF PROD-CODE = 0                      244    IF IND-DET = NEXT-DET
220    ELSE                                  247      PERFORM UPDATE-CUS-HIST
221      PERFORM READ-PROD-CODE.             252  UPDATE-CUS-HIST.
                                             263    IF REF-PURCH-STK(IND)
                                             264        = PROD-CODE
                                             267      MOVE PROD-CODE
                                             268       TO REF-PURCH-STK(IND)
```

FIGURE 149. The code fragment used to detect the referential constraint between CUS and STK.

The slice used to detect the second referential constraint (ORD → STK) is displayed in figure 148. The lines in italic are added to ease the understanding of the code fragment. The lines given by the program slicing are only those that are on the path (in the SDG) from the `read` to the (re)`write`. But these lines are not always enough to understand how the program works. For example, there is a dataflow edge between line 245 and 195 that does not belong to the same paragraph. To understand this dataflow, we need to know how these two paragraphs are connected. The SDG (figure 150.a) shows that three dataflow arcs and four control flow arcs are necessary to express the relation between the `read` instruction (line 226) and the `write` instruction (line 196). The path alone is not enough to be sure that there is a referential constraint. To validate the referential constraint, we must be sure that the value stored by the `write` (line 196) is the same as the value read by line 226. To perform this, we need to add the lines 215 and 225 to find a common dataflow between the two instructions. The common dataflow uses `PROD-CODE` (a working variable) that contains a valid value of `STK-CODE`.

The slice used to detect the third referential constraint (CUS → STK) is displayed in figure 149. The SDG (figure 150.b) shows that three dataflow arcs and eight control flow arcs are necessary to express the relation between the `read` instruction (line 226) and the `rewrite` instruction (line 201). To validate the referential constraint, we must be sure that the value stored by the `rewrite` statement (line 201) is the same as the value read by line 226. To perform this we need to add the lines 215 and 225 to find a common dataflow between the two instructions. The common dataflow uses `PROD-CODE` (a working variable) that contains a valid value of `STK-CODE`.



a) The SDG of code fragment of figure 148

b) The SDG of code fragment of figure 149

FIGURE 150. The two SDG.

To be able to create those two referential constraints we have to, first of all, refine *CUS-HIST* and *ORD-DETAIL*. This refinement is not defined by the program slicing, but usually some hints can be found in the fragment proposed by the program slicing.

```
208  READ CUSTOMER INVALID KEY
188  MOVE CUS-HIST TO LIST-PURCHASE.
263  IF REF-PURCH-STK(IND)
264      = PROD-CODE
266  ELSE
267    MOVE PROD-CODE
268          TO REF-PURCH-STK(IND)
199  MOVE LIST-PURCHASE
200           TO CUS-HIST.
201  REWRITE CUS
```

FIGURE 151. The code to validate the dependency between `CUS-HIST` and `LIST-PURCHASE`.

The analysis of these code fragments also allows to recover the exact cardinalities, local identifiers and set types of the arrays.

The last line of the table of figure 147 means that `CUS-HIST` is read and written. It can be noticed that the write instruction is the same as the one used by the third line. This last line does not give any new information except that the entity type is read, modified and `REWRITE`.

The referential constraints and the local identifiers of the multivalued attributes can also be validated through data analysis. In the manual case study, we have to write the COBOL program manually. In the automated version, there exists a *Voyager2* program, that generates the COBOL program. This *Voyager2* program (`gen_validator.oxo`) generates the COBOL program with respect to the logical schema. The program generated can be seen in annex B, section B.2.

9.2.2. Data structure conceptualization

To automate the data structure conceptualization, we can write a transformation script that automatically produces a conceptual schema. The conceptual schema produced by this script is not yet the final conceptual schema because some transformations cannot been automated, they require human intervention. For example, human intervention is needed to rename the objects to give more meaningful names and to decide to decompose an attribute because its component represents different domain concepts (e.g., the disaggregation of *DESC*).

The suggested method is to rename the objects, then to use the script and finally finish manually the normalization of the schema.

9.2.2.1. Meaningful name

The names of the collections are more meaningful than the corresponding entity types one, so the name of the entity types are substituted by the name of their collection.

```
1. EXTERN "auto_concept.oxo".remove_prefix(ATT_per_ET(2 N))
2. REMOVE(ALL_COLL())
3. REMOVE(ALL_KEY())
4. DISAGGREGATE(SUB_ATT_per_ATT(1 1))
5. DISAGGREGATE(
     SUB_ATT_per_ATT(1 N) and MAX_CARD_of_ATT(1 1) and REF_per_ATT(1 N))
6. ATT_into_ET_INST(
     SUB_ATT_per_ATT(1 N) and MAX_CARD_of_ATT(2 N) and REF_per_ATT(1 N))
7. REF_into_RT(ALL_REF())
8. ET_into_RT(ALL_ET())
```

FIGURE 152. A possible script to automate the conceptualization of a COBOL logical schema.

9.2.2.2. Transformation script

To automatically transform the schema, an *advanced global transformation* script has been created. An advanced global transformation script is a list of transformations to be applied on the schema of the form *<transformation>(<predicate>)*. Each transformation has a predicate as argument, this predicate is used to select the objects on which the transformation must be applied. Figure 152 shows a simple script to transform a COBOL logical schema into its corresponding raw conceptual schema:

1. *EXTERN "auto_concept.oxo".remove_prefix(ATT_per_ET(2 N))*

   This is not a built-in transformation but a call to an external (*Voyager2*) function. The function is the `remove_prefix` function of the `auto_concept.oxo` *Voyager2* program. The predicate used to select the object on which `remove_prefix` must be applied is *ATT_per_ET(2 N)*, i.e., the entity types that have more than one attributes (between 2 and N). The function `remove_prefix` removes the common prefix of the attributes of an entity type.

2. *REMOVE(ALL_COLL())*

   The predicate ALL_COLL() selects all the collections of the schema and REMOVE() removes all the selected objects, i.e., it removes all the collections of the schema.

3. *REMOVE(ALL_KEY())*

   Removes all the access key.

4. *DISAGGREGATE(SUB_ATT_per_ATT(1 1))*

   Disaggregates all the compound attributes that have only one component.

5. *DISAGGREGATE(*
   SUB_ATT_per_ATT(1 N) and MAX_CARD_of_ATT(1 1) and REF_per_ATT(1 N))

   Disaggregates the atomic compound attributes that have a component that is the origin of a referential constraint.

6. *ATT_into_ET_INST(*
   SUB_ATT_per_ATT(1 N) and MAX_CARD_of_ATT(2 N) and REF_per_ATT(1 N))

   Transforms into entity type (by instance representation) the compound multivalued attributes that contain a referential constraint.

7. *REF_into_RT(ALL_REF())*

   Transforms all the referential constraints into a relationship type.

8. *ET_into_RT(ALL_ET())*

   Transforms all the entity types (that can be transformed) into a relationship type.

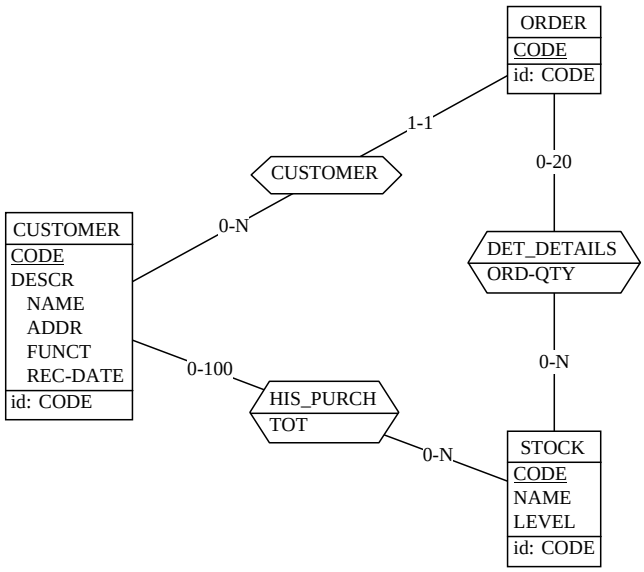


FIGURE 153. The conceptual schema obtained by the execution of the script of figure 152.

The execution of this script on the logical schema (figure 142) produces the conceptual schema (figure 153).

9.2.2.3. Normalization

The normalization must be done by hand to obtain the final conceptual schema.

A. Attribute disaggregation

The attribute `DESC` is composed of attributes of different domain concepts. We can decide to disaggregate them (**Transform - Attribute - Disaggregation**).

B. Name processing

Some names can be changed to be more meaningful. For example:

- The relationship type *DET_DETAILS* can be changed to *detail*.
- The relationship type *HIST_PURCH* can be changed to *purch*ase.
- The attribute *TOT* can be changed to *TOTAL*.
- The attribute *ADDR* can be changed to *ADDRESS*.
- The attribute *FUNCT* can be changed to *FUNCTION*.
- The attribute *ORD-QTY* can be changed to *QUANTITY.*

A usual naming rule is to use lowercase for relationship type names, uppercase for the entity type names and capitalized for the attribute names.

The final conceptual schema is shown in figure 145.

9.3. COBOL with embedded SQL

This case study will recover the complete logical schema and the conceptual schema of a SQL database. This fairly old database does not contain any explicit foreign keys. They are implicitly implemented in the COBOL program that accesses the data.

This example shows how to analyze a COBOL program with embedded SQL.

9.3.1. Project preparation

This project has three sources of information:

1. The SQL-DDL script that declares the database. This text declares the entity types (tables), attributes (columns), the identifiers (primary keys) and the access keys (indexes) but no foreign keys. The code can be found in annex B, section B.3.
2. The COBOL program, including embedded SQL statements, that add, modify and read data of the database. The code can be found in annex B, section B.4.
3. The populated database.

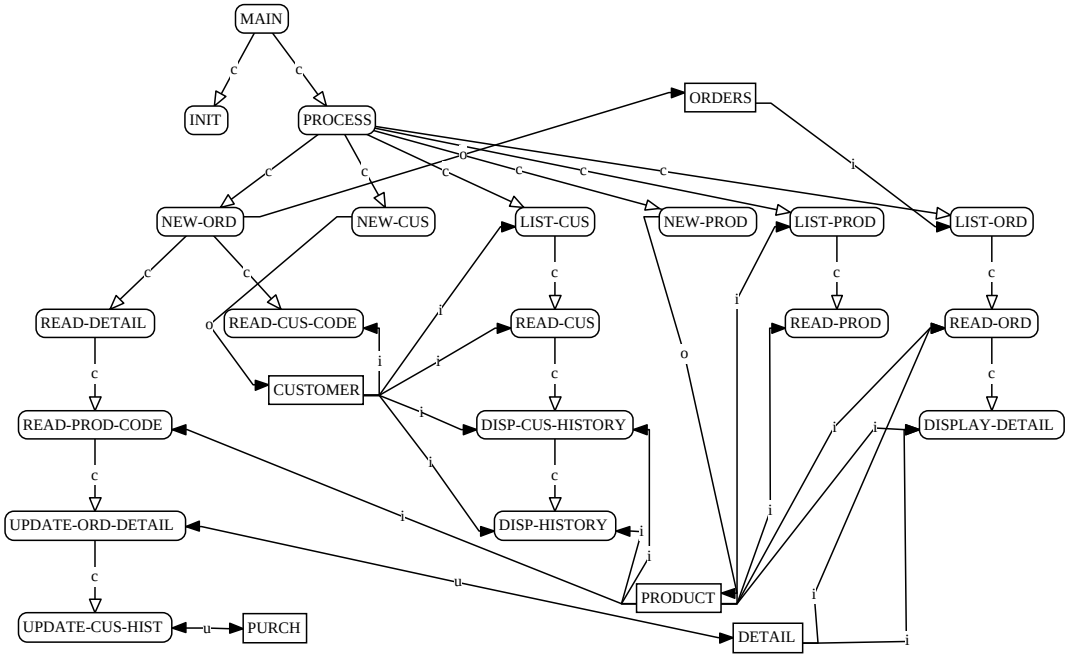


FIGURE 154. The procedure call graph and the data usage graph.

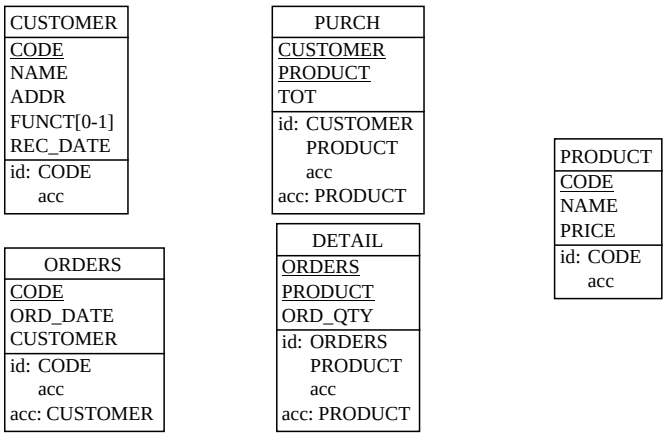


FIGURE 155. The raw physical schema.

The procedure call graph and the data usage graph are displayed in figure 154. In this graph the rounded rectangles represent the paragraphs (procedures) of the COBOL program, the "c" edges are the perform statements, the rectangles represent the tables and "i", "o", "u" represent the input (select), output (insert) and update (update or insert and select in the same paragraph) of a table by a paragraph. The call graph (the name of the paragraph and the "c" edges) is produced by the DB-MAIN program slicing tool. DB-MAIN does not offers a tool to retrieve the data usage graph, we have to write some grep/awk scripts to extract the table's name used into select, insert and update SQL statements to create the "i", "o" and "u" edges.

9.3.2. Data structure extraction

9.3.2.1. DDL code analysis

The SQL-DDL code (annex B, section B.4) is analyzed (**File - Extract - SQL**) to produce the raw physical schema (figure 155).

9.3.2.2. Schema refinement

In this example, schema refinement is limited to the recovery of the referential constraints.

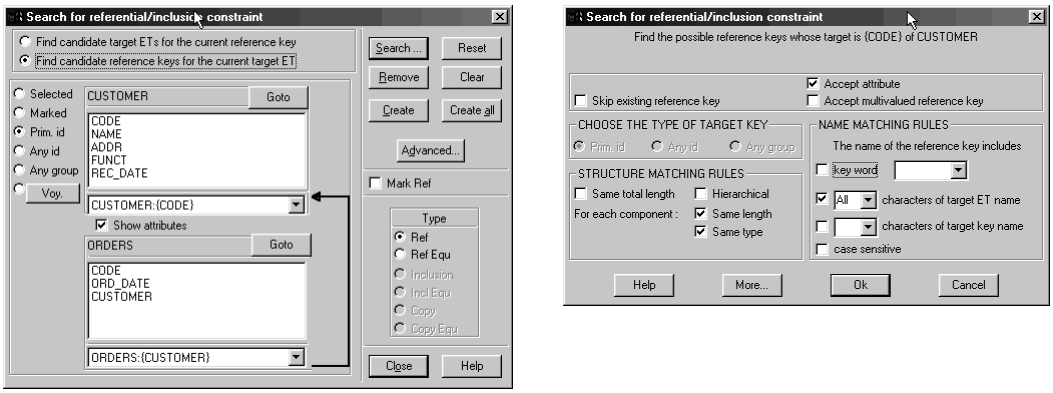


FIGURE 156. The configuration of the referential constraint assistant to discover the potential referential constraints.

Hypothesis discovery. The database was well designed and strict naming rules where used for entity types and attributes. To find the potential referential constraints, the analyst analyzes the physical schema with the reference constraints assistant. These referential constraints should target an identifier and the origin attribute should has the same name as the target entity type. Both sides should have the same length and same type. Figure 156 shows the configuration of the assistant to perform the search. It provides the following potential referential constraints:

- *ORDERS.CUSTOMER* → *CUSTOMER.CODE*
- *PURCH.CUSTOMER* → *CUSTOMER.CODE*
- *PURCH.PRODUCT* → *PRODUCT.CODE*
- *DETAIL.ORDERS* → *ORDERS.CODE*
- *DETAIL.PRODUCT* → *PRODUCT.CODE*

Hypothesis validation. To validate the proposed referential constraint, the analyst decides to analyze the program source code and to analyze the data.

There are two different ways to implement a join. The first one is to write a select query that contains a join. The second one is to implement a procedural join i.e., each SQL query (select, insert or update) queries only one entity type and the link between the entity types is done by some COBOL code.

```
194   EXEC SQL
195     DECLARE CUS_HIST CURSOR FOR
196     SELECT P.TOT, PR.NAME
197     FROM PURCH P, PRODUCT PR
198     WHERE P.PRODUCT = PR.CODE
199       AND P.CUSTOMER = :CUS-CODE
200    END-EXEC.
```

a) *PURCH - PRODUCT* join.

```
339    EXEC SQL
340     SELECT CODE
          INTO :PROD-CODE
341     FROM PRODUCT
342    END-EXEC.
       ....
357    EXEC SQL
358     SELECT ORD_QTY
359     FROM DETAIL
360     WHERE ORDER = :ORD-CODE
361       AND PRODUCT = :PROD-CODE
362    END-EXEC
```

b) *DETAIL - PRODUCT* join.

FIGURE 157. The two SQL joins found in the source code.

The source code contains two SQL joins (see figure 157). The analysis of these two joins increases the confidence in the two referential constraints that have *PRODUCT* as target. These two queries read and do not modify the data. To be sure that a constraint is present, each modification instruction must be analyzed to check if it is preceded by instructions that validate the constraint.

To analyze the procedural join with the program slicing technique, the program must be transformed to allow the construction of the SDG that represents the behavior of the COBOL instructions with the embedded SQL, as explained in 6.5. The transformed program can be found in annex B, section 2.5. In this program each, SQL query is commented (line prefixed by `*E`) and is replaced by its equivalent pseudo-instruction.

To recover the referential constraints, a program slice must be computed for each instruction that modifies (`insert` or `update`) the data.

```
283 NEW-ORD.                                  312 READ-CUS-CODE.
289    MOVE 1 TO SQLCODE.                     315    ACCEPT CUS-CODE.
290    PERFORM READ-CUS-CODE UNTIL SQLCODE =0.316*E    EXEC SQL
292*E    EXEC SQL                             317*E     SELECT CODE
293*E     INSERT INTO ORDERS VALUES(:ORD-CODE,318*E     INTO :CUS-CODE
294*E       SYSDATE(), :CUS-CODE)              319*E     FROM CUSTOMER
295*E    END-EXEC.                            320*E     WHERE CODE = :CUS-CODE
297   DIRECT-MAP CUS-CODE TO ORDERS--CUSTOMER.321*E    END-EXEC.
298    INDIRECT-MAP ORDER--CODE               322    DIRECT-MAP CUS-CODE TO CUSTOMER--CODE.
299      ORDERS--CUSTOMER TO SQLCODE.         323    INDIRECT-MAP CUSTOMER--CODE
                                              324      TO CUSTOMER--CODE SQLCODE.
                                              325    DIRECT-MAP CUSTOMER--CODE TO CUS-CODE.
```

FIGURE 158. Program slice with respect to line 298 and `ORDERS--CUSTOMER`.

There is only one query that inserts data into *ORDERS* (line 293). To verify that *ORDERS.CUSTOMER* is the origin of a referential constraint, a program slice is computed with respect to line 298 and `ORDERS--CUSTOMER` (see figure 158). This slice shows that the value of `ORDERS--CUSTOMER` (the variable representing *ORDERS.CUSTOMER*) is a valid value of `CUSTOMER--CODE` (the variable representing *CUSTOMER.CODE* the identifier of *CUSTOMER*). Thus the referential constraint is validated.

```
283 NEW-ORD.                                    338 READ-PROD-CODE.
287    ACCEPT ORD-CODE.                         339*E    EXEC SQL
289    MOVE 1 TO SQLCODE.                       340*E     SELECT CODE INTO :PROD-CODE
290    PERFORM READ-CUS-CODE UNTIL SQLCODE=0. 341*E     FROM PRODUCT
292*E    EXEC SQL                               342*E     WHERE CODE = :PROD-CODE
293*E     INSERT INTO ORDERS VALUES(:ORD-CODE,343*E    END-EXEC.
294*E       SYSDATE(), :CUS-CODE)                344    DIRECT-MAP PROD-CODE TO PRODUCT--CODE.
295*E    END-EXEC.                                345    INDIRECT-MAP PRODUCT--CODE
296    DIRECT-MAP ORD-CODE TO ORDERS--CODE.     346      TO PRODUCT--CODE SQLCODE.
297   DIRECT-MAP CUS-CODE TO ORDERS--CUSTOMER.347    DIRECT-MAP PRODUCT--CODE TO PROD-CODE.
298    INDIRECT-MAP ORDERS--CODE                348    IF SQLCODE = 0
299      ORDERS--CUSTOMER TO SQLCODE.           349      PERFORM UPDATE-ORD-DETAIL.
300    IF(SQLCODE NOT = 0)                      353 UPDATE-ORD-DETAIL.
302    ELSE                                     357*E    EXEC SQL
303      MOVE 0 TO END-FILE                     358*E     SELECT ORD_QTY
304      PERFORM READ-DETAIL.                   359*E     FROM DETAIL
312 READ-CUS-CODE.                              360*E     WHERE ORDER = :ORD-CODE
315    ACCEPT CUS-CODE.                         361*E        AND PRODUCT = :PROD-CODE
316*E    EXEC SQL                               362*E    END-EXEC
317*E     SELECT CODE                           363    DIRECT-MAP ORD-CODE TO DETAIL--ORDERS
318*E     INTO :CUS-CODE                        364    DIRECT-MAP PROD-CODE TO DETAIL--PRODUCT
319*E     FROM CUSTOMER                         365    INDIRECT-MAP DETAIL--ORDERS
320*E     WHERE CODE = :CUS-CODE                         DETAIL--PRODUCT
321*E    END-EXEC.                                366      TO DETAIL--ORD-QTY SQLCODE.
322    DIRECT-MAP CUS-CODE TO CUSTOMER--CODE. 367    IF SQLCODE = 0
323    INDIRECT-MAP CUSTOMER--CODE              369    ELSE
324      TO CUSTOMER--CODE SQLCODE.             370*E      EXEC SQL
325    DIRECT-MAP CUSTOMER--CODE TO CUS-CODE. 371*E       INSERT INTO DETAIL
330 READ-DETAIL.                                372*E         VALUES (:ORD-CODE, :PROD-CODE,
332    ACCEPT PROD-CODE.                              :DET-QTY)
333    IF PROD-CODE = 0                         373*E      END-EXEC
334      MOVE 1 TO END-FILE                     374      DIRECT-MAP ORD-CODE TO DETAIL--ORDERS
335    ELSE                                     377      INDIRECT-MAP DETAIL--ORDERS
336      PERFORM READ-PROD-CODE.                            DETAIL--PRODUCT
                                                378        DETAIL--ORD-QTY TO SQLCODE.
```

FIGURE 159. Program slice with respect to line 377 and `DETAIL--ORDERS`.

The query that inserts data into *DETAIL* is at the line 371. To validate the referential constraint (*DETAILS.ORDERS*, *ORDERS.CODE*), the program slice with respect to line 377 and `DETAIL--ORDERS` (the variable representing *DETAIL.ORDERS*) is computed (see figure 159). This slice shows that there is a referential constraint from *DETAILS.ORDERS* to *ORDERS.CODE*.

```
283 NEW-ORD.                                  338 READ-PROD-CODE.
287    ACCEPT ORD-CODE.                       339*E    EXEC SQL
289    MOVE 1 TO SQLCODE.                     340*E     SELECT CODE INTO :PROD-CODE
290    PERFORM READ-CUS-CODE UNTIL SQLCODE =0.341*E     FROM PRODUCT
292*E    EXEC SQL                             342*E     WHERE CODE = :PROD-CODE
293*E     INSERT INTO ORDERS VALUES(:ORD-CODE,343*E    END-EXEC.
294*E       SYSDATE(), :CUS-CODE)              344    DIRECT-MAP PROD-CODE TO PRODUCT--CODE.
295*E    END-EXEC.                             345    INDIRECT-MAP PRODUCT--CODE
296    DIRECT-MAP ORD-CODE TO ORDERS--CODE.    346      TO PRODUCT--CODE SQLCODE.
297   DIRECT-MAP CUS-CODE TO ORDERS--CUSTOMER.347    DIRECT-MAP PRODUCT--CODE TO PROD-CODE.
298    INDIRECT-MAP ORDERS--CODE              348    IF SQLCODE = 0
299       ORDERS--CUSTOMER TO SQLCODE.        349      PERFORM UPDATE-ORD-DETAIL
300    IF(SQLCODE NOT = 0)                    353 UPDATE-ORD-DETAIL.
302    ELSE                                   357*E    EXEC SQL
303      MOVE 0 TO END-FILE                   358*E     SELECT ORD_QTY
304      PERFORM READ-DETAIL .                359*E     FROM DETAIL
312 READ-CUS-CODE.                            360*E     WHERE ORDER = :ORD-CODE
315    ACCEPT CUS-CODE.                       361*E        AND PRODUCT = :PROD-CODE
316*E    EXEC SQL                             362*E    END-EXEC
317*E     SELECT CODE                         363    DIRECT-MAP ORD-CODE TO DETAIL--ORDERS
318*E     INTO :CUS-CODE                      364    DIRECT-MAP PROD-CODE TO DETAIL--PRODUCT
319*E     FROM CUSTOMER                       365    INDIRECT-MAP DETAIL--ORDERS
320*E     WHERE CODE = :CUS-CODE                       DETAIL--PRODUCT
321*E    END-EXEC.                                    366      TO DETAIL--ORD-QTY SQLCODE.
322    DIRECT-MAP CUS-CODE TO CUSTOMER--CODE. 367    IF SQLCODE = 0
323    INDIRECT-MAP CUSTOMER--CODE            369    ELSE
324      TO CUSTOMER--CODE SQLCODE.           370*E      EXEC SQL
325    DIRECT-MAP CUSTOMER--CODE TO CUS-CODE. 371*E       INSERT INTO DETAIL
330 READ-DETAIL.                              372*E  VALUES(:ORD-CODE,:PROD-CODE, :DET-QTY)
332    ACCEPT PROD-CODE.                      373*E      END-EXEC
333    IF PROD-CODE = 0                       375    DIRECT-MAP PROD-CODE TO DETAIL--PRODUCT
334      MOVE 1 TO END-FILE                   377      INDIRECT-MAP DETAIL--ORDERS
335    ELSE                                              DETAIL--PRODUCT
336      PERFORM READ-PROD-CODE.              378        DETAIL--ORD-QTY TO SQLCODE.
```

FIGURE 160. Program slice with respect to line 377 and `DETAIL--PRODUCT`.

To validate the referential constraint (*DETAIL.PRODUCT* → *PRODUCT.CODE*), the program slice is computed with respect to line 377 and `DETAIL--PRODUCT` (the variable representing *DETAIL.PRODUCT*), as shown in figure 160. This slice shows that there is a referential constraint from *DETAIL.PRODUCT* to *PRODUCT.CODE*.

```
404      EXEC SQL
405       UPDATE PURCH SET
406           TOT = (:PURCH-TOT + :DET-QTY)
407       WHERE CUSTOMER = :CUS-CODE
408         AND PRODUCT = :PROD-CODE
409      END-EXEC.
```

FIGURE 161. The query that updates *PURCH*.

```
283 NEW-ORD.                                    338 READ-PROD-CODE.
287    ACCEPT ORD-CODE.                         339*E    EXEC SQL
289    MOVE 1 TO SQLCODE.                       340*E     SELECT CODE INTO :PROD-CODE
290    PERFORM READ-CUS-CODE UNTIL SQLCODE =0.341*E     FROM PRODUCT
292*E    EXEC SQL                               342*E     WHERE CODE = :PROD-CODE
293*E     INSERT INTO ORDERS VALUES(:ORD-CODE,343*E    END-EXEC.
294*E       SYSDATE(), :CUS-CODE)                344    DIRECT-MAP PROD-CODE TO PRODUCT--CODE.
295*E    END-EXEC.                                345    INDIRECT-MAP PRODUCT--CODE
296    DIRECT-MAP ORD-CODE TO ORDERS--CODE.       346      TO PRODUCT--CODE SQLCODE.
297   DIRECT-MAP CUS-CODE TO ORDERS--CUSTOMER.347    DIRECT-MAP PRODUCT--CODE TO PROD-CODE.
298    INDIRECT-MAP ORDERS--CODE ORDERS--         348    IF SQLCODE = 0
     CUSTOMER                                   349      PERFORM UPDATE-ORD-DETAIL.
299      TO SQLCODE.                            353 UPDATE-ORD-DETAIL.
300    IF(SQLCODE NOT = 0)                      379      PERFORM UPDATE-CUS-HIST.
302    ELSE                                     381 UPDATE-CUS-HIST.
303      MOVE 0 TO END-FILE                     382*E    EXEC SQL
304      PERFORM READ-DETAIL                    383*E     SELECT TOT INTO :PURCH-TOT
312 READ-CUS-CODE.                              384*E     FROM PURCH
315    ACCEPT CUS-CODE.                         385*E     WHERE CUSTOMER = :CUS-CODE
316*E    EXEC SQL                               386*E         AND PRODUCT = :PROD-CODE
317*E     SELECT CODE                           387*E    END-EXEC.
318*E     INTO :CUS-CODE                        388    DIRECT-MAP CUS-CODE TO PURCH--CUSTOMER.
319*E     FROM CUSTOMER                         389    DIRECT-MAP PROD-CODE TO PURCH--PRODUCT.
320*E     WHERE CODE = :CUS-CODE                390    INDIRECT-MAP PURCH--CUSTOMER
321*E    END-EXEC.                              391      PURCH--PRODUCT TO PURCH--TOT SQLCODE.
322    DIRECT-MAP CUS-CODE TO CUSTOMER--CODE. 393    IF(SQLCODE = 0)
323    INDIRECT-MAP CUSTOMER--CODE            394*E      EXEC SQL
324      TO CUSTOMER--CODE SQLCODE.           395*E      INSERT INTO PURCH VALUES(:CUS-CODE,
325    DIRECT-MAP CUSTOMER--CODE TO CUS-CODE. 396*E                    :PROD-CODE, :DET-QTY)
330 READ-DETAIL.                               397*E      END-EXEC
332    ACCEPT PROD-CODE.                        398     DIRECT-MAP CUS-CODE TO PURCH--CUSTOMER
333    IF PROD-CODE = 0                         401      INDIRECT-MAP PURCH--CUSTOMER
334      MOVE 1 TO END-FILE                                PURCH--PRODUCT
335    ELSE                                     402        PURCH--TOT TO SQLCODE.
336      PERFORM READ-PROD-CODE.
```

FIGURE 162. Program slice with respect to line 401 and PURCH--CUSTOMER.

Two queries modify the values of the data of *PURCH*, namely the insert *PURCH* query at line 395 and the update query at line 405. The update query (see figure 161) only modifies the value of *TOT*, on which no constraint holds. To validate the referential constraint (*PURCH.CUSTOMER*, *CUSTOMER.CODE*), the program slice with respect to line 401 and PURCH--CUSTOMER (the variable representing *PURCH.CUSTOMER*) is computed (figure 162). This slice shows that there is a referential constraint from *PURCH.CUSTOMER* to *CUSTOMER.CODE*.

```
283 NEW-ORD.                                    338 READ-PROD-CODE.
287    ACCEPT ORD-CODE.                         339*E    EXEC SQL
289    MOVE 1 TO SQLCODE.                       340*E     SELECT CODE INTO :PROD-CODE
290    PERFORM READ-CUS-CODE UNTIL SQLCODE =0.  341*E     FROM PRODUCT
292*E    EXEC SQL                               342*E     WHERE CODE = :PROD-CODE
293*E     INSERT INTO ORDERS VALUES(:ORD-CODE,  343*E    END-EXEC.
294*E       SYSDATE(), :CUS-CODE)                344    DIRECT-MAP PROD-CODE TO PRODUCT--CODE.
295*E    END-EXEC.                               345    INDIRECT-MAP PRODUCT--CODE
296    DIRECT-MAP ORD-CODE TO ORDERS--CODE.     346      TO PRODUCT--CODE SQLCODE.
297   DIRECT-MAP CUS-CODE TO ORDERS--CUSTOMER.  347    DIRECT-MAP PRODUCT--CODE TO PROD-CODE.
298    INDIRECT-MAP ORDERS--CODE                348    IF SQLCODE = 0
299      ORDERS--CUSTOMER TO SQLCODE.           349      PERFORM UPDATE-ORD-DETAIL.
300    IF(SQLCODE NOT = 0)                      353 UPDATE-ORD-DETAIL.
302    ELSE                                     379      PERFORM UPDATE-CUS-HIST.
303      MOVE 0 TO END-FILE                     381 UPDATE-CUS-HIST.
304      PERFORM READ-DETAIL .                  382*E    EXEC SQL
312 READ-CUS-CODE.                             383*E     SELECT TOT INTO :PURCH-TOT
315    ACCEPT CUS-CODE.                         384*E     FROM PURCH
316*E    EXEC SQL                               385*E     WHERE CUSTOMER = :CUS-CODE
317*E     SELECT CODE                           386*E         AND PRODUCT = :PROD-CODE
318*E     INTO :CUS-CODE                        387*E    END-EXEC.
319*E     FROM CUSTOMER                         389    DIRECT-MAP PROD-CODE TO PURCH--PRODUCT.
320*E     WHERE CODE = :CUS-CODE                390    INDIRECT-MAP PURCH--CUSTOMER
321*E    END-EXEC.                              391      PURCH--PRODUCT TO PURCH--TOT SQLCODE.
322    DIRECT-MAP CUS-CODE TO CUSTOMER--CODE.   393    IF(SQLCODE = 0)
323    READ-MAP CUSTOMER--CODE                  394*E      EXEC SQL
324      TO CUSTOMER--CODE SQLCODE.             395*E       INSERT INTO PURCH VALUES(
325    DIRECT-MAP CUSTOMER--CODE TO CUS-CODE.   396*E         :CUS-CODE, :PROD-CODE, :DET-QTY)
330 READ-DETAIL.                               397*E      END-EXEC
332    ACCEPT PROD-CODE.                        399     DIRECT-MAP PROD-CODE TO PURCH--PRODUCT
333    IF PROD-CODE = 0                         401      INDIRECT-MAP PURCH--CUSTOMER
334      MOVE 1 TO END-FILE                     402      PURCH--PRODUCT PURCH--TOT TO SQLCODE.
335    ELSE
336      PERFORM READ-PROD-CODE.
```

FIGURE 163. Program slice with respect to line 401 and `PURCH--PRODUCT`.

To validate the referential constraint (*PURCH.PRODUCT*, *PRODUCT.CODE*), the program slice with respect to line 401 and `PURCH--PRODUCT` (the variable representing *PURCH.PRODUCT*) is computed (figure 163). This slice shows that there is a referential constraint from *PURCH.PRODUCT* to *PRODUCT.CODE*.

```
--- constraint from DETAIL to ORDERS
select count(*)
  from DETAIL, ORDERS
  where DETAIL.ORDERS = ORDERS.CODE;

--- constraint from DETAIL to PRODUCT
select count(*)
  from DETAIL, PRODUCT
  where DETAIL.PRODUCT = PRODUCT.CODE;

--- constraint from ORDERS to CUSTOMER
select count(*)
  from ORDERS, CUSTOMER
  where ORDERS.CUSTOMER = CUSTOMER.CODE;

--- constraint from PURCH to CUSTOMER
select count(*)
  from PURCH, CUSTOMER
  where PURCH.CUSTOMER = CUSTOMER.CODE;

--- constraint from PURCH to PRODUCT
select count(*)
  from PURCH, PRODUCT
  where PURCH.PRODUCT = PRODUCT.CODE;
```

FIGURE 164. SQL queries that validates the referential constraints.

Those five referential constraints can also be validated through data analysis. Such validation is quite easy thanks to the power of SQL. For each constraints a query is written that counts the number of the target entity type instances that violate the constraint. The queries that validate the five referential constraints are shown in figure 164.

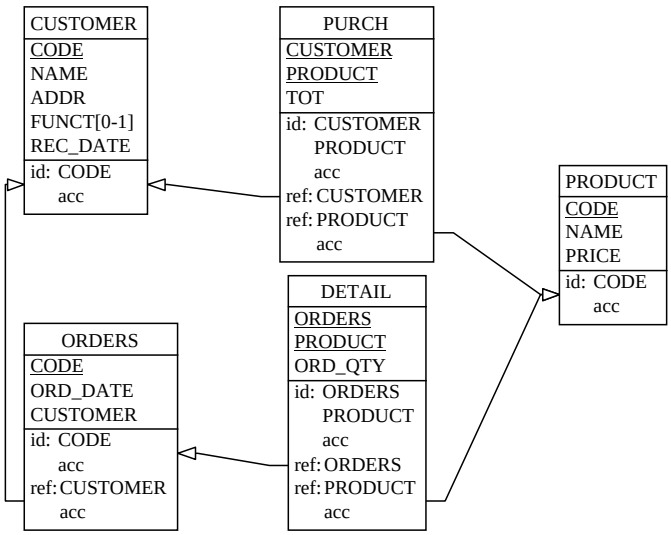


FIGURE 165. The complete physical schema.

Schema enhancement. Five referential constraints have been discovered and can be added to the physical schema to obtain the complete physical schema of figure 165.

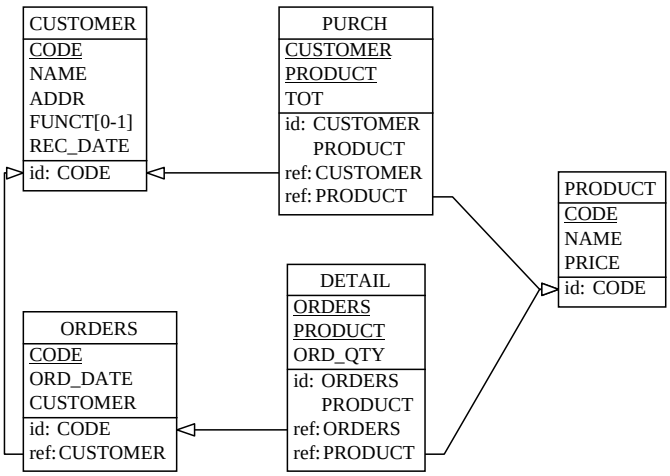


FIGURE 166. The complete logical schema.

9.3.2.3. Schema cleaning

The underlying database is a relational database, so that the access keys and the collections can be suppressed to obtain the complete logical schema (see figure 166).

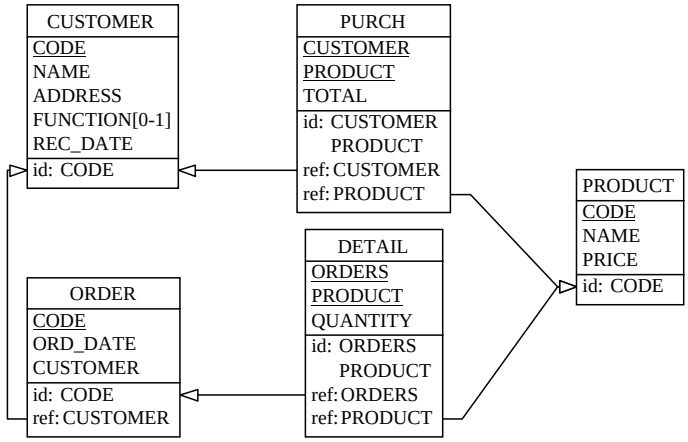


FIGURE 167. The prepared schema.

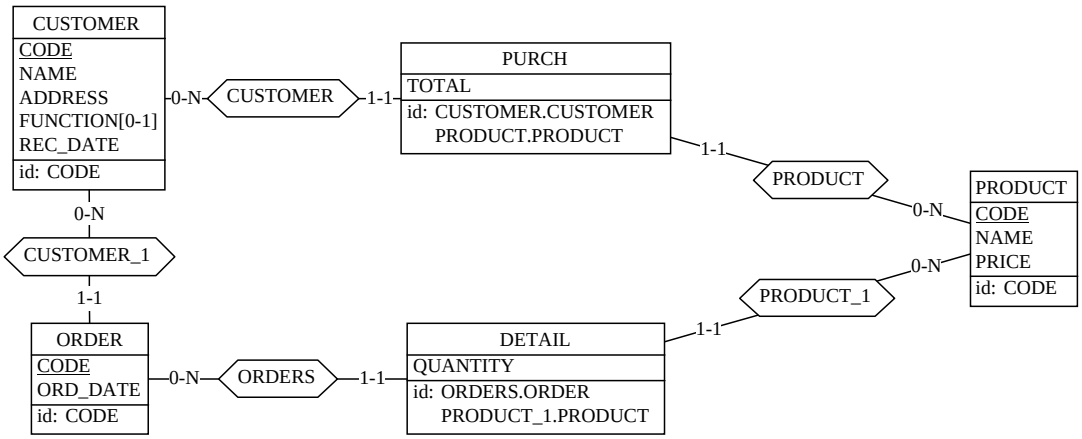


FIGURE 168. The raw conceptual schema.

9.3.3. Data structure conceptualization

9.3.3.1. Preparation

This phase prepares the schema such that it only contains structures and constraints that are necessary to understand the semantics of the schema.

This schema is already quite clean. Some names can be changed to provide more meaningful names or to comply with some naming convention. *ORDERS* is a plural and usually entity types have singular names. It is plural because `ORDER` is a SQL reserved word and the programmer decide to use the plural instead. *ORDERS* is changed to *ORDER*. The attributes *ADDR*, *FUNCT*, *REC-DATE*, *TOT* and *ORD-QTY* are abbreviations, they can be replaced with complete names (*ADDRESS*, *FUNCTION*, *RECORD-DATE*, *TOTAL*, *QUANTITY*).The prepared schema is shown in figure 167.

9.3.3.2. Basic conceptualization

During this step, all the foreign keys are transformed into relationship types (**Transform - Group - rel-type**) to provide the raw conceptual schema (see figure 168).

9.3.3.3. Normalization

During the normalization, the analyst gives to the schema the quality of a conceptual schema, i.e. readability, concision, minimality, expressiveness.

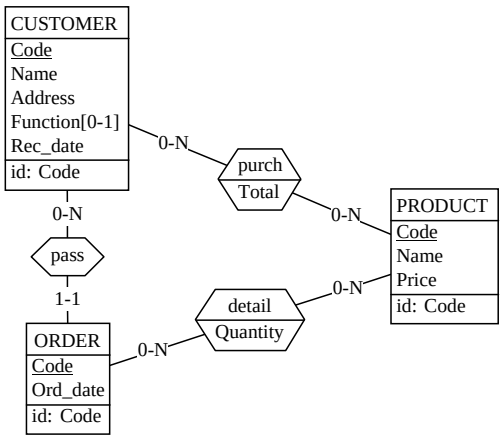


FIGURE 169. The conceptual schema.

The normalization is the same as in the two previous case studies. The final conceptual schema is shown in figure 169.

9.4. Real DBRE projects

In this section, we present real DBRE projects that we have carried out for companies. These projects were very valuable because they allowed us to understand the needs of the companies and the problems generated by the size of those projects. Moreover, they allowed us to validate our methodology and tools.

9.4.1. COBOL

9.4.1.1. Problem statement

The system analyzed in this case study was written in COBOL with indexed files in the late 1970's to manage most of the business of a quarry company: ordering, quarry, salary, warehouse, etc. The design of the original system was out sourced and the maintenance is supported by the only programmer of the company. There was no written documentation and the company wondered if it will buy a new package, maintain the existing one or develop a new system. To specify the new

system, they first have to know (or to document) the functionalities of the actual system. We were asked to help them to retrieve the logical schema of the current database.

The specific problems of this project were

- The files are declared in each program.
- The same physical file (on the disk) can have different logical name (`FD` paragraph in the `file section`).
- There were many aggregation files, i.e., files that contain statistics and aggregate values computed from other files.

9.4.1.2. Steps

In this project, we have started by the extraction of the files and records declarations. The declarations of each program were stored in a separate schema. The files were renamed to give them the name of the corresponding physical schema. This was needed to avoid to have two files in two programs that have the same name but do not correspond to the same physical file. The records were prefixed by the name of their file. This avoids to have two records with the same name belonging to two different files.

Next, all the schemas, one per program, were integrated in a new schema. The integration was done with respect to the name of the objects and a report was generated. The analysis of this report shows if two records with the same name had different structures. This step produced the integrated physical schema.

To retrieve the data dependencies, dataflow program slices were computed with respect to the `WRITE` and `REWRITE` instructions and the programmer was asked to validate the potential data dependencies discovered by the program slicing.

9.4.1.3. Results

The integrated physical schema was produced and the project was stopped because the programmer had no time to validate the data dependencies discovered by the program slicing.

9.4.1.4. Size

program		**database**		
KLOC	**programs**	**files**	**records**	**fields**
300	185	352	560	9150

9.4.1.5. Lessons learned

This project was the first real size project we conducted. It was very useful for us from a technical point of view, as well as from the point of view of project management.

Program understanding techniques were proved to be useful to retrieve implicit constructs. Even if the programmer knows the application, he is not able to remember each detail and he needs some support. This project shows that the analyst needs to master the program understanding techniques he uses and he must be able to interpret the results produced by the tools. Otherwise he can produce wrong schemas by an incorrect interpretation of the results.

It also highlights the necessity of tools and more precisely automatic tools to cope with the huge volume of the source code.

This project was also very interesting on the project management side. The training and information of the local team is a very important point for the success of such project. We spent a lot of time to explain (train) to the local team the expected results of program understanding and to "prove" that these results are correct.

We notice that the management of the company and the programmers must support the DBRE project. This project was more or less imposed by the management to the programmer and sometimes the programmer "didn't have the time", he did not understand the purpose of the project, etc.

9.4.2. ADS - IDMS

9.4.2.1. Problem statement

This system is used by a car importator to manage cars and trucks ordering in the different branches: list of the available vehicles models, list of the possible options for each model, expected delivery date, etc.

The system is written in ADS (COBOL like) and uses an IDMS database (CODASYL). At the origin there were six different applications, each application had its own database. These six applications were put together but the databases were not merged, gateways were written to propagate the modification from one database to the other.

The customer wanted to migrate this system to a new client-server system with a relational database. We were asked to retrieve the logical schema and to produce a conceptual schema (without all the redundancies).

The difficulties of this project were:

- Adapt the program slicing tool to the ADS language.
- Detect all the redundancies and the referential constraints.
- Produce a conceptual schema without the redundancies.

9.4.2.2. Steps

The first step was to extract the IDMS DDL to produce the physical schema. This step was easy because IDMS offers a single DDL that contains all the explicit constructs, thus we did not need the integration step.

To retrieve the data dependencies (referential constraints and redundancies), it was decided to use the dataflow program slicing with variables follow-up to produce the list of fields in relation. With the help of an analyst of the customer the list of field couples were qualified (referential constraints - 450, computed referential constraints - 50, redundancies - 550, business rules - 500, noise - 200). All the referential constraints, computed referential constraints and redundancies were added to the physical schema to produce the complete physical schema.

The physical schema was analyzed and transformed to produce a conceptual schema.

9.4.2.3. Results

We produced the logical schema and the conceptual schema. The conceptual schema was used to develop the new application.

9.4.2.4. Size

program		physical schema			conceptual schema		
KLOC	**modules**	**records**	**sets**	**fields**	**entity types**	**relationships**	**attributes**
200	170	324	380	2200	96	109	1100

- Number of program slices computed: 1000
- Number of dataflows found between a `get` and a `get` or a `store`: 5000

  A program slice computed w.r.t. a `get` or a `store` contains more that one `get` (5 in mean!).
- Number of data dependencies (number of fields couples): 1750
- Number of data dependencies validated: 1050

  i.e., the number of fields couples representing referential constraints, computed referential constraints, redundancies. These data dependencies have been validated by the local analysist.
- Total time : 62 days

9.4.2.5. Lessons learned

This project confirms the project management issues discovered in the previous one.

Program understanding proved to be very useful. When we presented the result, for validation, to the local programmer, he discovered or remembered relations between the data that would have been missed without program understanding techniques.

With this project, we proved that, by spending some time developing tools instead of doing the job manually, we can spare time and money in comparison to doing the job manually. To discover the data dependencies between the record types, we were obliged to modify the program slicing tool (10 days), then it took 3 days to analyze all the sources code with the tool and 15 days to analyze the results produced by the tool. So it took 28 days to analyze the 5000 dataflows to discover 1050 data dependencies.

We estimate the time necessary to discover and analyze, manually, one dataflow between a `get` and a `store` to 10 minutes. If we multiply this by 5000 (the number of dataflows), we obtain 120 days. So automatic program understanding tools enabled us to save 92 days on this project.

9.4.3. Centural / SQL

9.4.3.1. Problem statement

This system is used by a municipal administration. It is composed of six independent applications, each one used by a different service of the administration: civil status, population, taxes, etc. There was no communication between the different applications. For example, when somebody moves, he has to go in each service to give his new address. The municipal council has decided to set up the concept of "unique counter" where the citizen can go to perform all the usual administrative steps. To offer this service, the different applications must be merged.

All the applications were developed in the same language and with the same DMS: Centura, a 4GL and a relational database. There was no documentation of the application but the programmer who had developed the applications was still there for six months. We were asked to recover the logical schema of each application and to determine the common parts of the different databases.

Specific problems:

- There is no foreign key declared in the database.
- We don’t have analyzers for the 4GL.

9.4.3.2. Steps

The first step was to extract the DDL of each database. This step was quite easy because the SQL used by the DMS was standard. It produced a poor physical schema with very few identifiers declared, some indexes, no foreign keys and most of the columns were optional. This can be explained by the history of the applications that were migrated form an older relational DBMS.

The next step was to retrieve the referential constraints. We extracted the queries embedded in the code, but we noticed that there were very few joins in the queries. The queries were used to guess some of the identifiers. We decided to ask the programmer to add (manually) the referential constraints to the physical schema.

To validate the schema, queries were automatically generated. The results of the execution of the queries were used to correct the schema and to detect some erroneous data.

9.4.3.3. Results

The logical schema was produced as well as some assessments about the quality of the data. We noticed that there are very few common parts in the different databases. The only common part is the concept of "people".

9.4.3.4. Size

program		physical schema	
KLOC	**applications**	**tables**	**columns**
450	6	200	3500

9.4.3.5. Lessons learned

For this project, we did not have program understanding tools available and we estimated that their development was too expensive. The only program understanding technique we have tried is to analyze the embedded SQL queries.

We were surprised to notice that there is no join implemented in the SQL queries. All the joins were implemented procedurally and we were unable to discover them. This show that even if the DMS offers powerful techniques, the programmer does not necessarily use them. In this case, it can be explained by the history of the application. This application was translated from a COBOL program.

We asked the programmer to "remember" the missing referential constraints and to note them on the database physical schema. This schema was validated with respect to the data, but we are not sure that the programmer did not forget some of them.

9.4.4. IDEAL - Datacom-DB

9.4.4.1. Problem statement

This system is used by a company of mail order trading to manage all its business: orders, customers, invoices, advertisement, etc. The system is written in IDEAL (COBOL like) and uses a Datacom-DB database.

The company had maintained manually the conceptual schema of the database. But over the time, the conceptual schema had diverged from the physical database. Some modifications have been made on the structure of the database but were no transferred to the conceptual schema. We were asked to detect the differences between the conceptual schema and the physical schema.

Specific problems:

- Write a Datacom-DB extractor.
- Extract the conceptual schema from another CASE tool.
- Compare the two schemas.

9.4.4.2. Steps

We wrote a Datacom-DB extractor to be able to extract the physical schema. Next, we had written an extractor to import the conceptual schema into DB-MAIN. The conceptual schema was trans-

formed to obtain a physical version. The conceptual schema contains, as a description, the name of the corresponding object in the database, so we were able to rename the objects of the transformed schema.

To find the differences between both schemas, a program was written to compare both schemas based on the name of the objects. A report was produced with the differences between the schema and sent to the company's programmers. They were asked to solve the conflicts.

9.4.4.3. Results

A report with the differences between the physical schema and the conceptual schema was produced. We asked the programmers to resolve the conflicts, but they did not have time to perform this!

9.4.4.4. Size

program	**physical schema**		**conceptual schema**		
Modules	**records**	**fields**	**entity types**	**relationships**	**attributes**
4000	850	11500	556	812	3841

9.4.4.5. Lessons learned

This project shows that even when the documentation exists, if it is maintained manually, it will be slowly desynchronized with respect to the physical database. After some time (years) some DBRE is necessary to resynchronize the documentation with respect to the database.

CHAPTER 10 *DBRE project management issues*

Real size reverse engineering projects taught us that reverse engineering large databases with hundreds of programs can prove particularly difficult. Program understanding techniques are very powerful techniques that can be used to retrieve the implicit constraints. But these techniques are costly to use. The costs are of different types. They can be linked to the setup of the techniques and to the implementation of the tools that enact these techniques. The costs are linked to the labor too i.e. the analyst needs to master all these techniques, and therefore needs to be trained. Finally, the analysis of the program through program understanding techniques can take a lot of time. Even if these techniques are powerful and precise, the analyst could not guarantee the quality of the results. There is a risk that some constraints will not be discovered (silence) and some non-existing constraints will be stated (noise).

Those two aspects, cost and risk, of DBRE techniques, naturally lead to take into account project planning and management.

One of the unexpected lessons, learned with real projects, is that before the DBRE project itself can start, the DBRE team has to explain what DBRE is and to justify it. This evangelization must be done to convince the management but also the technical team. Management must be convinced that the organization is really going to achieve a significant benefit in reducing costs and adding value [Sneed-1995].

An important point in the planning of a DBRE project is to evaluate the costs of the project. This cost evaluation consists in the estimation of the tools, the skills and the time required by the project. It is impossible to give general rules to evaluate the cost of a project because each project is different and there are many factors (size, complexity, language, human factors, etc.) that influence these costs. Reducing the costs of a project can be achieved by only recovering some of the constraints and to make a lighter analysis. This incomplete analysis will produce a lower quality result (more silence and noise). Moreover lower quality schemas can induce costs later if the result of the DBRE is used as an input of another process. Another particularity of DBRE projects is that they are very difficult to evaluate because they produce an abstract result (the conceptual schema of the database) that can be used by other processes. The purpose of a DBRE project is to describe (document) an existing system. The customer is unable to validate the proposed schema of the database because he does not master the database, otherwise he does not need DBRE. The evaluation of a classical forward project is quite "simple", the customer can test the application and checks if it offers the

expected functionalities. There are two ways to evaluate a DBRE project. The first one is to redevelop a new application using the conceptual schema and to check if the new application has the same functionalities as the legacy one. This solution is very expensive and not always applicable. The second way is to explain to the customer the techniques that will be used and to explain the expected results. The customer evaluates the method instead of the results.

10.1. DBRE justification

Performing DBRE just to have an up-to-date documentation of the current application is not always (or rarely) a sufficient argument to convince a manager to give enough resources (human and budget) to conduct the project. DBRE is an expensive and risky operation that is not a goal by itself, but is a step in a larger process. DBRE is performed to ease or reduce the costs of maintenance, to regain control over the legacy system, to integrate existing systems, to migrate the legacy applications to a new hardware or software platform. Reverse engineering is in competition with the redevelopment of the system. Before starting a reverse engineering project, the whole process of migration or integration that includes reverse engineering, must prove to be less risky, less expensive, faster than the redevelopment of a new system. Application software systems may not have a uniform quality standard. Some programs may become obsolete before others, some may have a higher error rate, and others may be particularly difficult to reverse engineer. Therefore, it will not always be possible to judge on a system as a whole. It is possible to decide to reverse engineer some part of the legacy and to redevelop the other.

The strategy of just replacing the legacy applications by a new system may bring unexpected negative effects [Klösch-1996]. A lot of business rules have been coded into the legacy code and the knowledge of these rules can have disappeared. Nobody still knows these rules and the only place where they are formalized is in the source code. If a new system is developed, these rules need to be re-discovered or reinvented. People are accustomed with the functionalities of the legacy system. If the new one is too different, training may be necessary. The development of a new system is a long task and the result is uncertain. The new system may be out of date, more expensive than expected, it may contain errors or be less efficient than the legacy system.

A special argument in favor of DBRE is that the data are vital to the business of the company. For migration projects, it is easy to convince that DBRE is an obligatory step: while the application can be written from scratch, the data need to be migrated from the old to the new system. The data are the memory of the organization (orders, invoices, list of customers, etc.) so their structures need to be mastered to allow a correct migration.

If the semantics of the persistent data is well known, then it will be easier to understand the applications that use these data. The data and the applications can be migrated almost independently. If the database is well designed it is possible to keep the legacy database and to only migrate or rewrite the programs.

Another argument is that, if an organization masters its data and is convinced of their correctness, it may reduce its maintenance cost. If the structure of the data is well known, the maintenance team can evaluate which part of the system is affected by a modification and does not have the "surprise" to discover that some correct functions do not work any more after some maintenance processes. The maintenance can also be faster, because the maintenance team does not need to try to under-

stand the existing system before to start, it just have to find the relevant information in the system documentation.

Each interlocutor is sensitive to different arguments. Development managers and users would prefer to have applications rebuilt from scratch, corporate managers would like to purchase a commercial package and software maintainers usually prefer to keep things as they are [Sneed-1995].

10.2. Information / training

The maintenance and reverse engineering aspects of software engineering have been neglected by software engineers, researchers, schools and universities so far. But many companies are faced with a dilemma. On one hand, their systems are very valuable and simply replacing them may be too expensive. On the other hand, they do not understand them anymore, because the people who have developed the systems have left the company and the systems are becoming too expensive to maintain [Bennet-1995].

Companies are in the situation where they need reverse engineering and do reverse engineering without knowing that they are doing it and without any methods nor tools.

To successfully perform a reverse engineering project, a company must be aware of the difficulties and master the methods and techniques. Or at least, it must be conscious that it is not able to successfully perform the project and ask for external help.

Reverse engineering has been neglected by the research and academic community for different reasons. At first look, it does not seem very attractive to try to understand programs written many year ago by other programmers eventually using some old fashioned language. Researchers prefer to study new problems and use (or develop) new languages. It is easier to develop new programs from scratch than to analyze or modify existing programs. When a new application is developed from scratch, it is possible to decide which language to use, to setup rules on how to code the constraints, to elaborate naming conventions, etc. But when a researcher decides to define a reverse engineering methodology, he must take in account how the legacy was developed, in which language. The legacy is as it is, even if the program was badly written, the language is awful, etc. and he must understand it to recover the specification.

This explains why reverse engineering is not very popular. Companies do not know the term reverse engineering but they are doing it. So before to start a reverse engineering project, or even to convince a company it has to perform reverse engineering, we need to explain that reverse engineering is a known process for which there exist methods and techniques. We must convince that reverse engineering is not easy and it requires resource (budget), but it is possible.

The reverse engineer needs to make two categories of people aware of the very nature of reverse engineering: the managers and the technical team. Reverse engineering projects are risky and they do not add new functionalities to the applications. They increase the control on the applications, reduce the maintenance cost and allow smoother evolution of the legacy application to new technologies. To succeed in reverse engineering projects, the support of the management is an absolute necessity. To obtain this support, it must be made sensitive to the expected benefits. It needs to be

aware of the risk and costs of such a project. Under this condition the management is less reluctant to provide the necessary budget and to agree with programmers being assigned to the project.

The technical team (programmers, database administrators, etc.) will be the main contact point of the reverse engineer and a very valuable source of information. To allow a good collaboration between the reverse engineer and the local team, the local team must be convinced of the usefulness of the process, it must be allotted enough time to spend on this project and understand what reverse engineering is. The team must be made sensitive to the reverse engineering purpose and difficulties just like the management, but it also must master the technical aspects of the reverse engineering to use the same language and the same concepts as the reverse engineer.

Because reverse engineering is a quite new discipline, a lot of training efforts must be done before starting a project.

10.3. Project cost evaluation

A very important question, in real projects, is to evaluate the costs of a DBRE project using a given technique. This evaluation consists in the estimation of the time, the skills and the tools required by the project and in the planning.

Because each project is different, it is impossible to give a general method to evaluate its cost. This section will give a checklist of things that can influence the cost or need to be checked when evaluating a project.

- *Size of the legacy system*

  The most obvious parameter to predict the costs of a project is the size of the legacy system. There are many ways to measure this size (metric). A first measure is the size of the legacy database schema that can be evaluated in terms of number of entity types, collections, attributes per entity type, but also the link between entity types (relationship types or referential constraints). It is easy to understand that if a database has a lot of entity types, its reverse engineering will take a lot of time. But if there are a lot of referential constraints, it means that they have been explicitly declared and do not need to be recovered. So the costs will decrease.

  To recover the implicit constraints, the source code is analyzed, so some measures of the legacy code can be useful such as number LOC, function points, etc. [Mills-1988].

- *Complexity of the legacy system*

  The costs are directly related to the complexity of the legacy system. If a program has many function calls, go to's and tests, it will be more difficult to analyze than a program without go to's and only some test and function calls. This complexity can also be measured by some metrics such as the cyclomatic complexity or program knots [Mills-1988]. The complexity of a legacy system can also be evaluated by the usage of the entity types by the different modules. If an entity type is only used by one module, the program will be easier to analyze than a program in which almost every module accesses most entity types.

- *Information sources available*

  If there is an up-to-date documentation, it can be a good starting point and it can save a lot of time. On the other hand, if only the source code exists, program understanding techniques have to be used and this can take a lot of time.

- *Organization has its legacy system under control*

  The local team (the team that daily maintains the system) is very important in any DBRE project, because it is the contact point of the DBRE team. If the local team masters its applications, it can give all the information needed (no silence) and only them (no noise), so the DBRE team works on the right information. This can reduce the cost of the preparation process. The local team can answer precisely all questions asked by the DBRE team. For example, if the analyst suspects a referential constraint between two attributes, he can ask the local team to validate the hypothesis. This validation just takes a few seconds for somebody that masters the legacy system, but it may take several minutes, if the local team is unsure, and have to check in some documentation or in the legacy code.

- *Local team involvement*

  It is necessary that the local team participates in the project and it has to be convinced of the usefulness of the project. Even if it does not know each detail of the legacy application, it has a better knowledge of the application and of its domain than the reverse engineer does. Before starting the project it must be known if the team agrees to collaborate with the reverse engineer and if it has the time to do so.

- *Availability of tools*

  In some projects, new tools must be developed. These initial costs needs to be compared with the cost of performing the whole process by hand. For small projects it can be less expensive to do it by hand than to develop new tools.

- *Language used*

  Legacy system programming language and DMS used also influence the costs. For example, to reverse engineering a SQL database, there is only one source of information to analyze to have the complete physical schema. But if the legacy system uses COBOL files, the file declarations of each program need to be analyzed. The expressiveness of the DDL is also important, since it influences the explicit constraints that can be found. For example, during the analysis of a SQL database, the analyst can expect that all the referential constraints are explicitly declared and thus he does not have to perform complex program understanding to recover them.

- *Analyst knowledge of the programming language*

  To understand someone else's code, the analyst needs to have an in-depth understanding of the programming language. This can be problematic if the application was written in some old esoteric language. Quite often, those legacy languages are not taught anymore. The learning effort of the analyst needs to be evaluated.

- *Explicit declaration in DMS*

  The explicit declarations (DMS-DDL) are the starting point of the data structure extraction. So if a lot of the constraints have been declared in the DMS, there are fewer implicit constraints that need to be discovered in the procedural part of the application.

- *Quality of the code*

  To understand well structured code with well chosen variable names and procedure names is easier (and faster) than to analyze some ugly code with a lot of `go to`'s and obscure names.

- *Uniformity of the conventions / rules*

  If the same naming and coding conventions have been applied consistently during all the application live cycle and by all the programmers, the code will be easier to understand than if different conventions have been applied to each module or in the same module.

- *Analyst DBRE experience*

  Personnel quality is, according to Sneed [Sneed-1991], the most influential factor in driving maintenance cost. This also applies to reverse engineering. The analyst must be familiar with the methodology and the tools used. The comprehension of someone else's program requires some kind of feeling that cannot be learned in text books, but can only be acquired by experience.

- *Level of detail needed*

  Of course, the more details are needed, the more time it will take. If only the list of the entity types with their corresponding attributes is needed, it will only take a few hours. But to recover all the referential constraints and data dependencies, it will take days or months.

All these parameters are difficult to evaluate a priori, just using some metrics (as LOC, number of tables, etc.) and the list of the DMS and programming languages. A good solution is to start with a small representative sub-project. This project needs to be representative of the whole application. But it also needs to be small enough to be performed in a few days. To select this sub-project, the assistance of the local team is necessary to find a coherent sub-part of the application and representative of the coding style of the whole application. This sub-project can also be useful to set up, with the local team, some common language to communicate the questions, the answers and the results.

The main purpose of this sub-project is not to perform some DBRE, but to evaluate the feasibility and difficulties of the project. At the end of this sub-project, missing tools are listed, methodology to use is defined, the quality and weakness of the expected results are identified and some evaluation of the total cost is given.

Even if the prototype was successfully completed, the extrapolation of the total costs is difficult. During the project, unexpected difficulties can be discovered. To anticipate or minimize those planning and cost overtaking, it is important to plan checkpoints regularly with the local management.

Intuitively the complexity of DBRE projects is between $O(V)$ and $O(V^3)$, where V represents some measure of the size of the reverse engineering project. For instance, each entity type can be semantically related with any of the entity types of the schema. Therefore, each couple of entity types has to be examined to check if one or more referential constraints exist between them. So this leads to a complexity in square of the number of entity types in the database. In addition, the source code of the programs needs to be examined to validate those constraints. So the complexity of the DBRE projects rises with the square of the number of entity types and linearly with the number of the LOC. This leads to a process with complexity $O(V^3)$, where V is the size of the project, i.e. some measurement of the number of entity types in the database and the number of LOC, the DMS used, the complexity of the application, etc.

Fortunately, some structures require a lower complexity. For example, to refine the data structure, only the data dependency graph (which may be linear to the program size) needs to be computed. So the complexity is linear with respect to the number of LOC. Therefore that it can be stated that, considered as a whole, the data reverse engineering process has a complexity $O(V^2)$.

The cost of DBRE projects is in direct relation with its complexity, so it is also a function of the square of V. If such a cost evaluation function is applied to very big projects, they become unfeasible. This first evaluation must be revised because there is an effect of training. At the beginning of the project the analyst takes time to understand some aspects of the application and the more the project progresses the faster the analyst understands new details. As it will be explained in the next

section, for big projects, some processes can be automated to reduce the time spent by the analyst analyzing the application.

10.4. Automation

DBRE principles and methodology presented so far are well understood and can be quite easily applied to small and well structured systems. But when the methodology is applied to real size, complex systems, the analyst is faced with a huge volume of complex information to be manipulated and analyzed.

When analyzing a very small legacy system (e.g., 3 programs totaling 1000 LOC and 6 entity types), the database schema can be drawn on a single sheet of paper (or a single computer screen) and flip manually through the source code. This project can be easily completed in one or two days. The analyst can discover and remember most of the application details without any tool support.

The analysis of a medium size legacy system (200 programs totaling 400000 LOC and 100 entity types with an average of 60 attributes per entity type) requires a square meter sheet to draw the schema and it is impossible to flip through all the source code (more than 6000 pages). To draw the schema, at least, drawing tools are needed to ease the layout of the schema. To work efficiently, a CASE tool is useful to extract the schema from the DDL and give some help to correctly display the entity types to minimize their overlapping. To refine the physical schema through the analysis of the code, some program understanding tools are needed to help discovering the implicit constraints.

So automation is highly desirable to perform large DBRE projects within reasonable time and cost limits. It is usually admitted that an analyst can manipulate (manually) 50000 LOC [Tilley-1998], but real projects can be ten to hundred times this size.

The automation of the process does not mean that the complete DBRE process will be done automatically without the analyst's intervention. Instead, in most processes, the analyst is provided with tools that help him in his work. He has to decide which tool he wants to use at a given time and how to interpret the results. Many of the tools are not intended to locate and find implicit constructs, but rather contribute to the discovery of these constructs by focusing the analyst's attention on fragments of code or structural patterns or to aid the analyst to acquire a better understanding of the application. In short, they narrow the scope of search. It is up to the analyst to decide if the constraint that he is looking for is present or not. For example, computing a program slice provides a small set of statements with a high density of interesting patterns according to the construct that is searched for (typically referential constraints or attributes decomposition). This small program segment must then be examined visually (manually) to check whether traces of the construct are present or not.

All the steps of all the projects cannot be automated to the same degree. Different levels of automation can be enumerated: complete automation, automation with some interaction with the analyst, report generation or restriction of the search space.

A process that is *completely automated* is a process for which there exists a tool that takes as input the source code, the DDL or the incomplete database schema and enriches this schema with new constraints. The new schema contains all the constraints that the tool searches for without any inter-

vention of the user. For example, the DDL extractors automate the DDL extraction. They read the DDL code and automatically produce the corresponding structure of the schema.

Some processes can be *partially automated* with some interaction with the analyst. As in the fully automated one, the inputs are the source code, the DDL or the partial schema and the tools ask questions to the analyst to make some choices. For example, the dataflow diagram can automatically detect the actual decomposition of an attribute. The analyst is involved in conflict resolution (e.g., when two different decomposition patterns for the same attribute, he has to decide which one to use).

There exist tools that *generate reports* so that the analyst can analyze them to detect the existence of a constraint. For example, a report can be generated that contains couples of attributes in relation. A couple (*a*,*b*) means that there is a dataflow from attribute *a* to attribute *b*. If a dataflow exists between two attributes, this means that there is a potential referential constraint between those attributes, or a functional dependency or one is a function of the other or there is a business rule that involves both of them. The analyst has to read the report to find from this list which are referential constraints. To perform this selection, he has to use other techniques such as his knowledge of the domain.

Search space restriction tools are used to extract a fragment from a source of information that contains the pertinent information in which the analyst is supposed to find evidence to prove or disprove the existence of a constraint. Program slicing is a good example of search space restriction. When a slice is computed for an instruction, *x*, with respect to a variable, *v*, only the instructions that influence the value of *v* at *x* are selected. So the analyst can concentrate his effort on those lines only.

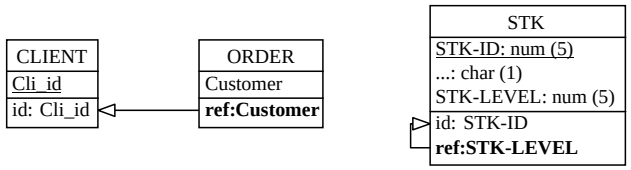
10.4.1.Limits of automation

A first reason why full automation cannot be reached is that a DBRE process basically is a decision-based activity. The discovery of referential constraints in a program source code is an example of constraint elicitation that cannot be fully automated. For example, to discover a referential constraint through program understanding techniques, data dependencies between the attributes of two entity types are searched for. However data dependencies do not necessarily materialize a referential constraint. A functional dependency may hold between the two attributes (such as a price and the price with VAT) or it can code some business rules (such as the order number is some function of the customer number and the order date). So, though one can imagine a tool that finds data dependencies between database attributes, the analyst still needs to qualify those dependencies to mark the one that actually represents a referential constraint.

Another reason for which full automation cannot be reached is that each DBRE project is different. The sources of information, the underlying DMS and the programming language or the coding rules can all be different and even incompatible. Designing a unique tool that will perform the complete DBRE for any project is unrealistic. The analyst must be provided with a set of tools in which he can select the one he needs and these tools must be configured for the current project. For example, referential constraints can be recovered through the analysis of the length, type and names of the attributes, entity types of the physical schema. Programmers can use different naming conventions to name the referential attributes. The name of the referential attribute contains (or suggests) the name of the target attribute, the name of the target entity type, or other rules that can be imagined. In some projects there are no rules at all and the analysis of the physical schema is useless. Each

project requires some specific tools to discover automatically some constraints. Another example is the attribute declaration in SQL-DDL. As will be explained in the next section, it can be decided not to construct such tools for economic reason and to perform the task manually.

Even for activities of the DBRE process that can be partially or completely automated, the tools must be used with some precaution [Wilde-1990]. While tools are likely to provide better results than unaided hand analysis, the analyst needs to understand how the tools produce the result. There are still many cases in which tools either fail to capture some constraints (missed targets or silence) or show constraints that do not really exist (false targets or noise). The analyst must validate the results and it is his responsibility to accept or not the constraints proposed by the tools. Automation can produce less precise results or incomplete results. The analyst must be aware of the potential errors contained in the produced result.



a) Referential constraint not discovered by the referential constraint assistant.

b) Referential constraint erroneously discovered by the referential constraint assistant.

FIGURE 170. Example of silence and noise generated by the referential constraint assistant that could be avoided through a manual analysis.

Even if some tools can help the analyst, the tools alone do not automatically lead to increased productivity. Training and well defined methods are needed to get a real benefit. Data structure extraction is a difficult and complex task that requires skilled and trained analysts who know the languages, DMS and operating system used by the legacy system and also master DBRE methodology, techniques and tools. Figure 170.a shows such an example; the origin attribute is *Customer* and the target entity type is *CLIENT*, the tool does not find this constraint, which can be discovered by manual analysis (synonym). This tool can also produce noise. For example (figure 170.b), a usual habit of COBOL programmers is to prefix all the attributes of an entity type by the name of the entity type (to have unique names). If an attribute has the same type and same length as the identifier of the entity type, it will be selected as a possible referential constraint.

10.4.2.Economic advantage of automation

As previously said, automation can influence the time spent (and cost) to complete a project. It is interesting to evaluate the cost evolution depending on the project size and the level of automation.

Even in manual projects, a small part of the job can be done by some existing tools, such as DDL extractors. The analyst masters these tools and did not need to customize them, so he can immediately be productive. Most of the job is done manually by analyzing the schema and the source code. Small project can be done by only one analyst and he can memorized most of the details. When the size of the project increase more tan one analyst are needed and they could not memorized all the details. In addition to the reverse engineering work, some team management is needed. For very big projects the manual approach can be inconceivable. The time needed to perform them are to long.

For example, a reverse engineering project that will last more than ten years is useless, which is the utility of such a project?

In automatic projects, where most of the work is done automatically, the initial costs can be quite high. The specificity of the project has to be understood to select the right tools, how to use them, how o interpret their results and to develop new ones. When the initial phase id one, the major part of the job can be done automatically with almost no intervention of the analyst. In the automatic approach, the manual part consists in the analysis and validation of the results produced by the tools.

Because of the initial cost of the automatic approach, for small to medium project the manual approach can be less expensive. But when the size of the project increases, the initial cost of the automatic approach can be compensated by the automation of the whole process

As an example, considering a very small application (one program of 1000 LOC and 3 entity types). An analyst can recover the complete schema of this application manually in one or two days. For a medium size application (200 programs of 400K LOC and 100 entity types), it is very difficult to carry out this project manually. Each program can not be analyzed independently. To analyze this application, the analyst has to follow the inter-program calls. If he takes only one hour to discover each referential constraint or functional dependency and there are 1000 constraints to discover, he will take about 40 days to complete the project. To perform this project automatically, the analyst needs 10 days to adapt program understanding tools. Such tools will analyze all the programs in three hours and then the analyst will validate the results in 10 days.

10.5. Cost Vs. quality

Cost reduction can be achieved by automation, but also by reducing the completeness of the results, e.g., by deciding not to search for some kind of constraints. For example exact cardinalities of multi-valued attributes or attribute decompositions may be ignored. The completeness can also be reduced by accepting less precise results, with more silence and more noise. For example, finding attribute decomposition can be done by computing the dependency graph and searching this graph for attributes that are connected (directly or indirectly) to a variable with a more precise decomposition. When a new decomposition is found, no other verification (such as code analysis) is done and the schema is considered refined. As explained in section 6.3 such techniques can produce erroneous decompositions and miss other decompositions.

Noise can easily be detected and suppressed by checking through another technique (such as data analysis or domain knowledge). Silences are more difficult to detect.

The overall quality of the DBRE is higher if the depth of analysis and checking is homogeneous throughout the schemas. To ensure the homogeneous quality, the analyst needs to apply the selected techniques and tools to all the sources of information. For example, lets assume he only analyzes 75% of the source code to recover the referential constraints. If the referential constraint validations are homogeneously distributed in the source code, 25% of the referential constraints potentially discovered by the analysis technique are missing. It is impossible to know which part of the schema is affected by the missing constraints. Are they all in the same region of the schema or are they randomly distributed? In some favorable situations the 25% non analyzed code does not contain

any new constraints, they have all been discovered by the analysis of the 75% of the code. This situation may arise if the analyst knows the code and decides to skip some parts of the code because these parts do not reveal new constraints.

DBRE generally is the first step of a broader process such as re-engineering or migration. Its costs must be evaluated on the whole process and not on each step separately. The results of the DBRE process are the input of another process. So if these results (logical and conceptual schemas) are of poor quality, the next process can have very high unexpected costs because its starting point is not correct. For example, if a new function is added to an existing application using a database schema in which some referential constraints are missing, this function can corrupt the database by adding or modifying data that violate the referential constraint. Missing constraints leave the database unprotected against data corruption. As a consequence, existing functions could not work anymore because their code is relying on correct data. For example, if a function prints a report and tries to find the target of an undetected referential constraint, a missing target can produce unexpected results (unexpected program termination, report with random data, etc.). Noise can also produce errors, since spurious constraints can prevent some valid data to be added.

Fixing these errors can have very high costs. Correcting the logical schema and the conceptual schema produced by the DBRE is only a small part of the cost. These schemas have been used to write new functions (or applications), so that all the newly developed code must be corrected to be in accordance with the new constraints. If the errors are detected only when the new application is used in production (with real data) the corrupted data must be also corrected. Sometimes the errors can cause the failure of a migration project.

This shows the importance to find the right trade off between the DBRE cost and the quality of the result.

10.6. DBRE project evaluation

The evaluation of the results produced by a DBRE project is a complex problem for the customer, especially when the DBRE is an explicit process in a project or when it is a project by itself.

Classical software engineering projects produce concrete results that are reasonably easy to evaluate, i.e., some piece of software that the user can use and test to check if it meets the initial specifications, notably through standardized testing techniques. The specificity of DBRE projects is that the final results are abstract specifications, made up of logical and conceptual schemas. The input of a reverse engineering project, which corresponds to the forward engineering specification, is the application itself [Henrard et al.-2000]. The customer has no deterministic means to evaluate the results except his domain knowledge and his (partial) knowledge of the application. This knowledge is partial by nature, otherwise DBRE would have been useless and thus the customer could have recovered the logical schema by himself [Chiang et al.-1996].

Several techniques can be proposed to help the customer in assessing the quality of the results.

1. One way to convince the customer that the DBRE is complete and correct would be to use DBRE results to migrate the application to a new database that is derived from the conceptual schema and to check if the new application has the same behavior as the old one. This solution requires to migrate the entire application (and thus reverse engineer) too or to write a new appli-

cation. This approach does not only test the quality of the DBRE process, but also the quality of the new application. If the behavior of the new application is not the same as the old one, it can be due to errors in the underling database. This a posteriori approach is only realistic when the DBRE is a first step in a migration process and the same team carries out the whole process.

2. Another way to validate the schema a posteriori is to automatically generate a new application, to migrate the data and to ask the users of the legacy application to test this new application. This new application does not intend to replace the legacy application, it does not offer all its functionality, but is simply used to validate the DBRE process. The structure of the new database can be easily generated from the conceptual schema. The knowledge of the legacy database and of the new database physical schema allows to automatically migrating the data [Henrard et al.-2002]. The migration is a first validation of the conceptual schema. If the data are correct in the legacy database and they could not be migrated to the new database, it means that there is an error in the conceptual schema. From the conceptual schema it is possible to generate automatically a graphical user interface (GUI) to access and to modify the data. Of course, it is about a rudimentary version of the legacy application, but it could be enough for a user to access and modify the data. Some users are asked to use this prototype to check if they could edit valid data and if the application forbids to enter erroneous data.
3. A realistic approach could be to agree, at the beginning of the project, on the methodology and the tools, i.e., on the constraints that are looked for and the techniques and tools used. It is also important to explain to the customer the strengths and the weaknesses of the chosen approach so that he can evaluate the quality of the expected results.

The critical process with respect to the quality of DBRE and thus the most important to evaluate, is the data structure extraction. Indeed, the quality of the conceptual schema depends mainly on the quality of the logical schema because it is obtained by semantic preserving transformations. The quality of the data structure extraction depends on the analyst's skill and tools, on the quality of the information sources but also on the constraints the analyst is looking for and on the time spent during the analysis.

Not surprisingly, the quality of the results is thus an economic issue.

CHAPTER 11 *Conclusion*

This thesis explores the role of program understanding techniques in database reverse engineering. The particularity of this approach is that it combines theories from two communities, namely software and database. Although these communities are historically distinct, but nevertheless intersect quite often. We have combined that effort from both areas to produce a improved database reverse engineering framework. The database community has developed well formalized models to represent data structures and to manipulate these structures. The software engineering community offers a large choice of techniques and tools to analyze code, a process that has proved necessary to understand data structures.

A generic DBRE methodology is presented as the reverse of forward engineering. This methodology is divided into three processes. The *project preparation* process evaluates why and how the project has to be carried out and which resources (human skills, time, budget, etc.) are needed. The second process, *data structure extraction*, analyzes all available sources of information to recover the complete logical schema of the data. This schema is the view the application programmers have (or should have) of the database to correctly develop and maintain the programs that access or modify the data. The last process, *data structure conceptualization*, transforms the logical schema into the conceptual schema.

The data structure extraction process has been presented in detail and particularly its most important, but also most difficult, sub-process, namely the *schema refinement* step. Schema refinement is the step during which the implicit constraints are recovered. Its difficulty comes from the fact that a wide variety of heterogeneous information sources (source code, data, user knowledge, etc.) have to be analyzed to recover the implicit constraints, i.e., constraints that hold in the data structure, but have not been declared in the DDL of the DMS. Another difficulty is that there is no standard way to express the implicit constraints. Therefore, in order to discover a constraint, the analyst first has to discover (e.g., to guess) how the programmers usually coded it. The importance of schema refinement can be appreciated by considering that, in most legacy databases, the largest part of the structures and constraints are implicit, either due to the weaknesses of the DMS, or to programming practices resorting to so-called *information hiding*. Therefore, recovering database physical and logical schemas through mere DDL code parsing generally leads to a highly incomplete result, generally useless for any decent further use. For instance, extracting a conceptual schema, from this logical schema, then generating a new relational database will provide the community with a particularly poor database as far as expressiveness and integrity are concerned.

We have presented most of the constraints that can be recovered during schema refinement, the information sources that can be analyzed and the heuristics that are used to recover the constraints. An application that updates data must check that the new data do not violate the constraints. There are mainly two ways for an application to validate the data it stores in the database: either the DMS carries out the validation and only accepts valid data or some code added to the application's procedural sections to validate the data before they are stored. In this second approach, unfortunately very popular, the implicit constraints are validated by source code fragments that have to be identified, through source code analysis techniques, generally called *program understanding*.

The assertion that program understanding is necessary to perform DBRE may be felt in contradiction with the justification of DBRE in the beginning of the thesis, where we stated that program understanding is easier when the database's structures and constraints have been elicited. There is no real contradiction. When the source code is analyzed to recover data structure constraints, we do not try to understand how the program works but we are only looking for code fragments that validate data structure constraints. To recover these fragments, we use program understanding techniques such as variable dependency graph, system dependency graph and program slicing. These techniques have been specialized to DBRE and we have shown how they can be used to detect the most popular constraints. Program understanding as a whole is a wider-scope discipline into which we have, hopefully, modestly contributed by exploring one part of the application, namely the database.

Real size reverse engineering projects have taught us that recovering the schemas of large databases can prove particularly long and difficult, and therefore risky and expensive. In order to cope with these difficulties, we have discussed some project management aspects. Because DBRE is expensive and, as such, does not add new functionalities to the application[1], it is important for such projects to be supported by the company's managers and not only by the technical team. In order to catch the confidence of all stakeholders, we have first to explain what DBRE actually is and what the expected results are. Indeed, the concept of DBRE (and reverse engineering in general) is not correctly known, and often leads to unrealistic expectations. Many people practice DBRE in an informal way, but they generally do not know about the existence of specific methodologies and tools to support this process. Manual work is the rule, leading to poor, incomplete and frustrating results.

To prove that DBRE is feasible and useful, we would need to evaluate *a priori* the cost of such a project. This evaluation has to take in account many parameters, such as the size of the database schema, the size of the programs, the complexity of the application, the programming language, the DMS, etc. In order to reduce the costs of a DBRE project, some of the processes must be automated. Another way to reduce the costs of a project is to reduce its quality requirements. This approach must be adopted with much caution, because the quality of the process can have an indirect cost impact on the following processes that use the result of DBRE as input. In a typical migration project, any error in the conceptual schema will generally induce errors in the new database, in the programs and in the functionalities that will be very expensive to fix.

1. Though it does add value.

11.1. Contributions

In this thesis, we have proved that program understanding techniques and tools are necessary to perform good quality DBRE for real size projects. To achieve this objectif, we have developped a tool supported DBRE methodology.

With regard to methodology, we have particularlydeveloped the schema refinement step of the data structure extraction process. We have proposed guidelines to carry out schema refinement as an iterative process during which the analyst makes assumptions about a potential constraint (the hypothesis) before validating this hypothesis. If the hypothesis is validated, the constraint is added to the schema. This process is iterated until no new hypotheses can be formulated. We have discussed how the analyst can decide whether the schema refinement is finished.

The starting point of the thesis is the idea that the source code is the most complete and the most up-to-date source of information to recover implicit constraints. Indeed, they must be dealt with in the source code because, by definition, they are ignored by the DMS. One of the exceptions are the constraints that are verified by the environment of the application (environmental properties) and therefore need not be verified formally. The source code is up-to-date because it is the readable expression of the programs currently running. Other sources of information, such as the documentation or CASE tools repositories, are often out-of-date due to the lack of direct link between them and the binary code.

This is why we have decided to investigate source code analysis to refine the database schema. The difficulty with code analysis is that the constraint validation can be spread through the whole source code and a given constraint is not necessarily coded in adjacent lines of code. A further difficulty related to constraint recovery is that there is not only one manner to code a given constraint, each programmer has his own way of coding. We have shown that, for example, there exist at least five different ways to validate a referential constraint.

Our experience has shown that code analysis can be practiced manually for small case studies but that medium to large projects require sophisticated tool support. In order to provide this support, we have studied program understanding techniques and implemented them in the DB-MAIN CASE tool. We have selected four techniques, namely pattern searching, variable dependency graph, program slicing and system dependency graph.

The variable dependency graph is a graph in which each vertex represents a variable and (directed) edges represent a relation between variables. The slice of a program with respect to program point *p* and variable *v* consists of all the program statements that might affect the state of *v* at point *p*. The system dependency graph is the internal representation of the program that is used to compute a program slice. We have defined other ways to question this graph to obtain very precise information about how the data are validated before being stored into the database. We have explained how these program understanding techniques can be used to recover the most popular implicit constraints. A major part of our work was to develop a tool box that supports the complete DBRE process. After describing the characteristics of an ideal CARE tool, we have presented our tools implemented in DB-MAIN.

In order to validate our methodology and the tools developed, we have used them to reverse engineer real databases in several companies. During those projects, we had to face problems that have nothing to do with methodology but rather with project management. For instance, client personnel

training has proved a major, though inexpensive, success factor. In this way, gaining the confidence, and therefore the collaboration of the persons who hold critical information is much easier.

The cost evaluation of such a project is crucial. We have sketched what the factors are that can influence the costs, among them the level of the process automation.

11.2. Comparison with related work

11.2.1.Methodology

We can identify three major periods in DBRE research, according to the information sources taken into account and to the initial assumptions on the quality of the object database.

11.2.1.1. Structural analysis

The first DBRE methodologies that appeared in the literature intended the production of a conceptual schema of an existing database by analyzing the database itself only. The main information source was the DDL code of the physical schema and in some contribution the database contents. For additional refinement, they relied on the analyst's domain knowledge. The quality of the final conceptual schema depended on the quality of the physical schema, that had to meet strong, and therefore unrealistic, requirements.

For example, the schema had to be in 3NF ([Navathe et al.-1987], [Davis et al.-1987], [Johannesson-1994], etc.). The identifier had to be declared ([Dumpala et al.- 1983], [Casanova et al.-1983]) or the dependencies between the records had to be known ([Dumpala et al.- 1983], [Casanova et al.-1983]). The database could not be optimized ([Ramanathan et al.-1996]). In some proposals, implicit foreign keys could be detected through the name of their components ([Navathe et al.-1987], [Premerlani et al.-1993], [Chiang-1995]).

Most of these methodologies were dedicated to a specific DMS, except MeRCI ([Comyn et al.-1996]). They only cover a part of the DBRE process.

11.2.1.2. Targeted code analysis

Two contributions only include program source code analysis to get information about the database structures and constraints.

[Petit-1996] analyzes the (embedded) SQL queries and views to recover the foreign keys and the functional dependencies. It is assumed that the programmer needs to use joins in the queries to navigate among the data. The joins are used to refine the logical schema that will be translated into a conceptual schema.

[Anderson-1996] analyzes COBOL source code to recover the precise structure of records (structure resolution) and uses some kind of dataflow analysis (the definition-use orders) to recover the dependencies between the records.

11.2.1.3. General code analysis

The two previous works concentrate on some aspects of the source code. In this thesis, we have increased the role of source code to make it a major information source, by integrating program understanding techniques in DBRE and more specifically program slicing and its underlying SDG. The SDG construction is difficult and expensive but produces precise results with little silence and noise. It produces better results than the definition-use order graph because SDG also represents control flow that was not taken into account.

11.2.2. Tools

Many methodologies are not tool-supported. Some of them suggest such tools ([Dumpala et al.-1983], [Signore et al.-1994], [Ramanathan et al.-1996]). Others are accompanied by some kind of *proof-of-concept* prototypes that support a part to the methodology ([Premerlani et al.-1993], [Chiang-1995], [Jahnke-1999]). These prototypes are most often intended to validate the methodology on case studies, few of them have been used to in a significant number of real size projects.

The main advantage of the tools we have developed is that they are integrated in the DB-MAIN CASE tool. DB-MAIN is a general database engineering CASE tool that supports not only the whole DBRE process, but also the main database engineering processes. For instance, an analyst can perform a database reengineeing project in a single environment.

DB-MAIN is a stable CASE tool that is being maintained by the Database Engineering Laboratory for more than 10 years and has several thousands of users. Those of its components that are dedicated to DBRE will survive this thesis and will be maintained and extended in the future.

11.2.3. Validation

Thanks to the robustness and completeness of the methodology and its CASE tool, it was possible to use them to validate the methodology on real projects in companies. It is important to note that the latter did not intend to merely *collaborate* with a research laboratory, but actually were our *client*. Therefore, the results were tested in real commercial conditions.

11.3. Future work

We have already got some experience in applying our methodology and tools to small to medium projects but we still have to validate them on larger projects (several million lines of code). In order to do so the level of automation of our tools must be increased and their results must be more precise to reduce the amount of work the analyst has to provide.

Some of our tools are specific to a given language such as program slicing or DDL extractors. Others are more generic such as the variable dependency graph that is parametrized by the patterns that define the nodes and edges of the graph in source programs. As expected, a tool that is dedicated to a language will grasp more semantics of the programs and provide more precise information. More research is needed to explore the way specialized tools can be developed more easily and

integrated into the DB-MAIN CASE tool. For example, automatically producing a program slicer by providing the language's grammar and some hints about the semantics of the main statements and data types should be possible.

Other PU techniques need to be analyzed and adapted to the need of DBRE to enrich the set of tools available. These techniques can improve existing techniques to produce more precise results, with less noise and less silence, such as dynamic program slicing. It will be, also, interesting to analyze the usage of techniques that give other information about the programs. Type inferencing is one of them. This technique groups the variables of a program according their values sharing / semantics.

Though it is one of the most challenging problems in reverse engineering, result validation was only sketched in this thesis. More investigation is needed to offer the analyst criteria to assess the extent to which the implicit structures and constraints have been recovered. This means in particular defining the ending criteria of the schema refinement process. Another aim of validation techniques is to convince the customer that the results of the DBRE process is of good quality and that they can be used in a subsequent project such as data migration, reengineering or maintenance. We have discussed three solutions: to carry out the complete migration, to develop a prototype to be tested by the user or to explain the methodology used and to convince the customer that it will produce (or has produced) a good result. Other solutions have still to be investigated.

The thesis tackles only DBRE, but as mentioned earlier it is only a step in a larger process such as data migration or reengineering. It could be interesting to study how our methodology can be coupled with those other processes to provide a complete solution for customers. In particular, the efforts made to understand the data structure aspects of the programs should be reused to understand the programs themselves. In addition, we wrote that database reverse engineering should be a good starting point to understand the whole application, but this assertion certainly deserves being further developed to provide better program understanding techniques. This thesis has shown that the database community can profit from the software engineering realm. The converse must obviously be true, but has still to be explored: how can database structure understanding contribute to a better understanding of data-centered application programs? Tool integration is also a major issue. Since no single tool can cope with all the aspects of system understanding and reverse engineering, different independent tools must have to cooperate. This leads to the problem of building adequate ontologies and exchange formats for this engineering domain.

Acronyms

ACFG	Augmented Control Flow Graph
AST	Abstract syntax Tree
B2B	Business to Business
CARE	Computed Aided Reverse Engineering
CASE	Computed Aided Software Engineering
CFG	Control Flow Graph
CORBA	Common Object Request Broker Architecture
DBD	IMS data description
DBMS	DataBase Management System
DBRE	DataBase Reverse Engineering
DDL	see DMS-DDL
DML	see DMS-DML
DMS	Data Management System
DMS-DDL	Data Management System - Data Description Language
DMS-DML	Data Management System - Data Manipulation Language
ECR	Entity-Category-Relationship
EER	Extended Entity-Relationship
ERP	Enterprise Resource Planning
FMS	File Management System
ODMG	Object Data Management Group
OMT	Object Modeling Technique
OO	Object Oriented
ORDBMS	Object Relational DataBase Management System
PDG	Procedure Dependency graph

PDL	Pattern Definition Language (DB-MAIN)
PS	Program Slicing
PSB	IMS program specification block
PU	Program Understanding
RDBMS	Relational DataBase Management System
SDG	System Dependency Graph
UML	Unified Modeling Language
VDG	Variable Dependency graph (DB-MAIN)
XML	Extensible Markup Language

References

Agrawal et al.-1991 Agrawal H., DeMillo R.A.: Dynamic slicing in the presence of unconstrained pointers. In *Proc. of the ACM symposium on Testing and Verification*, 1991.

Aho et al.-1989 Aho A., Sethi R., Ullman J.: *Compilateur : Principes, Techniques et Outils*. Intereditions, 1989.

Akoka et al.-1998 Akoka J., Comyn-Wattiau, I.: MeRCI: An Expert System for Software Reverse Engineering. In *Proc. of the 4th World Congress on Expert System*, Mexico, 1998.

Akoka et al.-1999 Akoka J., Comyn-Wattiau I.: Rétro-conception des Datawarehouses et des Systèmes Multidimensionnels. In *Proc. of the INFORSID'99*, pages 227-255, France, 1999. INFORSID.

Alhaij et al.-2001 Alhajj R., Polat F.: Reengineering Relational Databases to Object-Oriented: Constructing the Class Hierarchy and Migrating the Data. In *Proc. of the 8th Working Conference on Reverse Engineering (WCRE'2001)*, pages 335-344, Germany. 2001, IEEE Computer Society.

Anderson-1996 Anderson M.: *Reverse Engineering of Legacy Systems: From Value-Based to Object-Based Models*. PhD thesis, EPFL, Switzerland, 1996.

Ball et al.-1992 Ball T. and Horwitz S.: Slicing Programs with Arbitrary Control Flow. Technical report tr1128, University of Wisconsin, 1992. ftp://ftp.cs.wisc.edu/tech-reports/reports/92/tr1128.ps.Z.

Batini et al.-1992 Batini C., Ceri S., Navathe S.: *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

Batini et al.-1993 Batini C., Di Battista G., Santucci G.: Structuring Primitives for a Dictionary of Entity Relationship Data Schemas. *IEEE TSE*, 19(4), 1993.

Baxter-1997 Baxter I., Mehlich M.: Reverse Engineering is Reverse Forward Engineering. In *Proc. of 4th Working Conference on Reverse Engineering (WCRE'97), The Netherlands*, 1997. IEEE computer Society Press.

Bennet-1995 Bennett K.: Legacy Systems: Coping with Success. *IEEE Software*, 12(1):19-23, 1995.

Binkley et al.-1996 Binkley D., Gallagher K.B.: Program slicing, Technical report Loyola College in Maryland, 1996. http://www.cs.loyola.edu/~kbg/survey.ps.gz.

Blaha et al.-1995 Blaha M.R., Premerlani W.J.: Observed Idiosyncracies of Relational Database Designs. In *Proc. of the 2nd Working Conf. on Reverse Engineering (WCRE'95)*, Toronto, July 1995. IEEE Computer Socity Press.

Blaha-1996 Blaha M.: A Catalog of Object Model Transformations, in *Proc. of the 3rd Working Conf. on Reverse Engineering (WCRE'96)*, Monterey, 1996. IEEE computer Society Press.

Bolois et al.-1994 Bolois G., Robillard P.: Transformations in Reengineering Techniques. In *Proc. of the 4th Reengineering Forum "Reengineering in Practice*", Victoria, Canada, 1994.

Brodie et al.-1995 Brodie M.L., Stonebraker, M.: *Migrating legacy systems. Gateways, Interfaces and the incremental approach*. Morgan, 1995.

Casanova et al.-1983 Casanova M.A., Amaral de Sa J.E.: Designing Entity-Relationship Schemes for Conventional Information Systems. In P.P. Chen, editor, *Proc. of the Internatial Conference on Entity-Relationship Approach (ER'83)*, pages 265-277, 1983.

Casanova et al.-1984 Casanova M.A., Amaral De Sa.: Mapping uninterpreted Schemes into Entity-Relationship diagrams: two applications to conceptual schema design. *IBM J. Res. & Develop.*, 28(1), 1984.

Chiang-1995 Chiang R.H.L: A Knowledge-Based System for Performing Reverse engineering of Relational Databases. *Decision Support Systems*, 13:295-312, 1995.

Chiang et al.-1996 Chiang R.H.L., Barron T., Storey, V.: A framework for the design and evaluation of database reverse engineering methods. *Data & Knowledge Engineering*, 21(1) 57-77, 1996.

Chikofsky-1990 Chikofsky E.J., Cross II J.H.: Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 13, 1990.

Choi et al.-1993 Choi J.-D., Burke M., Carini P.: Efficient flow-sensitive interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proc. of Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 223-245, 1993.

Choi et al.-1994 Choi J.-D., Ferrante, J.: Static slicing in the presence of goto statements. *ACM Transaction of Programming Languages and Systems*, 16(4), 1994.

Comyn et al.-1996 Comyn-Wattiau I., Akoka J.: Reverse Engineering of Relational Database Physical Schema. In *Proc. of the International Entity-Relationship Conference (ER'96)*, pages 372-391, Germany, 1996.

Corbi-1989 Corbi T.A.: Program Understanding: Challenge for the 1990s. *IBM System Journal*, 28(2), 1989.

D'Atri et al.-1984 D'Atri A., Sacca D.: Equivalence and Mapping of Database Schemes. In *Proc. of the 10th VLDB Conf.*, Singapore, 1984.

Davis et al. - 1985 Davis K., Arora A.K.: A Methodology for Translating a Conventional File System into an Entity-Relationship Model. In *Proc. of the 4th International Conference on Entity-Relationship Approach (ER'85)*, pages 148-159. IEEE Computer

Society and North-Holland, 1985.

Davis et al.-1987 Davis K., Arora A.: Converting a Relational Database Model into an Entity-Relationship Model. In Salvatore T. March, editor, *Proc. of the 6th Internatial Conference on Entity-Relationship Approach (ER'87)*, pages 271-285, 1987.

De Troyer-1993 De Troyer O.: *On data schema transformation*. PhD Thesis, University of Tilburg, Tilburg, The Netherlands, 1993.

Delcroix et al.-2001 Delcroix C., Thiran Ph., Hainaut J.-L.: Approche Transformationnelle de la Réingénierie des Données. Ingénierie des Systèmes d'Information (Réingénierie des données et des documents sur le web), 6(1), 2001.

Delvaux-1996 Delvaux P.: Volume: Estimation des Volumes. Technical report, computer science department, University of Namur, Belgium, 1996.

Detienne et al.-2001 Detienne V., Hainaut, J.-L.: CASE Tool Support for Temporal Database Design. In *Proc of ER'2001*, Yokohama, Japan, 2001. Springer-Verlag.

Dumpala et al.- 1983 Dumpala S.R. Arora S.K.: Schema Translation using the Entity-Relationship Approach. In *Proc. of the Internatial Conference on Entity-Relationship Approach (ER'83)*, pages 337-356, 1983.

Englebert-2000 Englebert V.: Voyager 2 (version 6.0) - Reference Manual. Technical report, Computer Science Departement, University of Namur, Belgium, 2000.

Fahrner et al.-1995 Fahrner C., Vossen G.: A survey of database design transformations based on the Entity-Relationship model. *Data Knowledge Engineering*, 15(3), 1995.

Ferrante et al.-1987 Ferrante J., Ottenstein K., Warren J.: The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319--349, July, 1987.

Fong et al. - 1993 Fong J., Ho M.: Knowledge-Based Approach for Abstracting Hierarchical and Network Schema Semantics. In *Proc. on the 12th International Conference on the Entity-Relationship Approach (ER'93)*, USA, 1993

Garcia et al.-1995 Garcia-Solaco M., Saltor F., Castellanos M.A: Structure Based Schema Integration Methodology. In *Proceedings of the 11th International Conference of Interoperable Database Systems*, IEEE CS Press, pp. 505-512, 1995

Hainaut-1981 Hainaut J.-L.: Theoretical and Practical Tools for Data Base Design. In *Proc. of the Very Large Data Bases, 7th International Conference*, pages 216-224, France, 1981. IEEE Computer Society.

Hainaut-1989 Hainaut J.-L.: A Generic Entity-Relationship Model. In *Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an In-depth Analysis*, 1989. North-Holland.

Hainaut-1991 Hainaut J.-L.: Database Reverse Engineering, Models, Techniques and Strategies. In *Proc of the 10th Conf. on Entity-Relationship Approach (ER'91)*, USA, 1991.

Hainaut et al.-1993a Hainaut J.-L., Chandelon M., Tonneau C. and Joris M.: Contribution to a Theory of Database Reverse Engineering. In *Proc. of the Working Conference on Reverse Engineering (WCRE'93)*, Baltimore, 1993. IEEE Computer Society Press.

Hainaut et al.-1993b Hainuat J.-L., Chandelon M., Tonneau C., Joris M.: Transformational Techniques for Database Reverse Engineering. In *Proc. of the 12th International Conf. on ER Approach*, Arlington-Dallas, LNCS. E/R instuture and Springer-Verlag, 1993.

Hainaut et al.-1994 Hainaut J.-L., Englebert V., Henrard J., HickJ.-M., Roland D.: Evolution of Database Applications: The DB-MAIN Approach. In *Proc. of the 13th Int. Conf. on ER Approach (ER'94)*, Manchester, 1994. Springer-Verlag.

Hainaut et al.-1995 Hainaut J.-L., Englebert V., Henrard J., Hick J.-M., Roland D.: Requirements for Information System Reverse Engineering Support. In *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering (WCRE'95)*, Toronto, July, 1995. IEEE Computer Society Press.

Hainaut et al.-1996a Hainaut J.-L.: Specification Preservation in Schema Transformations: Application to Semantics and Statistics. *Data & Knowledge Engineering*, Elsevier Science, 19:99-134, 1996.

Hainaut et al.-1996b Hainaut J.-L., Roland D., Hick J-M., Henrard J. and Englebert V.: Database Reverse Engineering: from Requirements to CARE Tools. *Journal of Automated Software Engineering*, 3(1), 1996.

Hainaut et al.-1996c Hainaut J.-L., Hick J.-M., Englebert V., Henrard J., Roland D.: Understanding Implementation of IS-A Relations. In *Proc. of the 15th Conf. on ER Approach (ER'96)*, Cottbus, 1996. Springer-Verlag.

Hainaut-1997a Hainaut J.-L., Hick J.-M., Henrard J., Englebert V., Roland D.: The Concept of Foreign key in Reverse Engineering: A Pragmatic Interpretative Taxonomy. Technical report, Computer Science Departement, University of Namur, Belgium, 1997.

Hainaut et al.-1997b Hainaut J.-L., Englebert V., Hick J.-M., Henrard J., Roland D.: Knowledge Transfer in Database Reverse Engineering - A Supporting Case Study. In P*roc. of the 4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, Amsterdam, The Netherlands, 1997. IEEE Computer Society Press.

Hainaut et al.-1997c Hainaut J.-L., Henrard J., Hick J.-M., Roland D., Englebert V.: Contribution to the Reverse Engineering of OO Applications - Methodology and Case Study. In *Proc. of the IFIP 2.6 WC on Database Semantics (DS-7)*, Leysin, Switzerland, 1997. Chapman-Hall.

Halpin-1995 Halpin T.A., Proper H.A: Database Schema Transformation and Optimization. In *Proc. of the 14th Int. Conf. on ER/OO Modelling (ER'95), 1995*.

Henrard et al.-1998a Henrard J., Englebert V., Hick J.-M., Roland D. , Hainaut, J.-L.: Program understanding in databases reverse engineering. In *Proc. of DEXA'98*, Vienna, 1998.

Henrard et al.-1998b Henrard J., Roland D., Englebert V., Hick J.-M., Hainaut J.-L.: Outils d'analyse de programmes pour la rétro-conception de bases de données. In *Actes du Xème Congrès INFORSID*, Montpellier, 1998.

Henrard et al.-1999 Henrard J., Hainaut J.-L., Hick J.-M., Roland D., Englebert V.: Data structure extraction in database reverse engineering. In *Proc. REIS'99 Workshop (ER'99)*, Springer Verlag, LNCS 1727, 1999.

Henrard et al.-2000 Henrard J., Hainaut J.-L., Hick J.-M., Roland D., Englebert, V.: From Micro-

Analytical Method to Mass Processing - The Economic Challenge. In *Proc. of the Workshop on Data Reverse Engineering (DRE'2000)*, Zurich, Switzerland, 2000.

Henrard et al.-2001 Henrard J., Hainaut J.-L.: Data dependency elicitation in database reverse engineering. In *Proc. of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*, Portugal, 2001. IEEE Computer Society Press.

Henrard et al.-2002 Henrard J., Hick, J.-M., Thiran, Ph., Hainaut, J.-L.: Strategies for Data Reengineering. In *Proc. of the 9th Working Conference on Reverse Engineering (WCRE'02)*, Richmond, 2002. IEEE Computer Society Press.

Hick-2000 Hick J.-M.: DB-MAIN Project: Transformations. Technical report, Computer Science Departement, University of Namur, Belgium, 2000.

Hick-2001 Hick J.-M.: *Evolution d'Applications de Bases de Données Relationnelles: Méthodes et Outils*. PhD thesis, Computer Science Departement, University of Namur, Belgium, 2001.

Horwitz et al.-1990 Horwitz S., Reps T., and Binkley D.: Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26--60, January, 1990.

IBM-1998 IBM: The Year 2000 and 2-Digit Dates: A Guide for Planning and Implementation. Technical report GC28-1251-08, 1998.

Jahnke et al.-1999 Jahnke J.-H., Wadsack J.P.: Varlet: Human-Centered Tool Support for Database Reengineering. In Ebert J., Kulllbach B., Lehner, F., editor, *Proc. of Workshop on Software-Reengineering*, Germany, 1999.

Jahnke-1999 Jahnke J.-H.: *Managing Uncertainty and Inconsistency in Database Reengineering Processes*. PhD Thesis, University of Paderborn, Germany, 1999.

Jajodia et al.-1983 Jajodia S., Ng P., Springsteel F.: The problem of Equivalence for Entity-Relatinship Diagrams. *IEEE Transaction on Software Engineering*, 9(5), 1983.

Johannesson-1994 Johannesson P.: A Method for Transforming Relational Schemas into Conceptual Schemas, in *Proc. of the 10th Int. Conf. on Data Engineering*, USA, pages 190-201. IEEE Computer Society, 1994.

Joris et al.-1992 Joris M., Van Hoe R., Hainaut J.-L., Chandelon M., Tonneau C., Bodart F. et al.: PHENIX: Methods and Tools for Database Reverse Engineering. In *Proc 5th International Conf. on Software Engineering and Applications*, Toulouse, 1992. EC2 Publish.

Klösch-1996 Klösch R.: Reverse Engineering: Why and How to Reverse Engineer Software. In *Proc. of the California Software Symposium (CSS'96)*, 1996.

Kobayashi-1986 Kobayashi I.: Losslessness and Semantic Correctness of Database Schema Transformation: Another Look of Schema Equivalence. *Information Systems*, 11(1):41-59, 1986.

Korel et al.-1988 Korel B., Laski J.: Dynamic Program Slicing. *Information Processing Letters*, 29(3): 155-163, 1988.

Landi et al.-1992 Landi W., Ryder B.: A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proc. of the 1992 ACM Conference on Programming Language*

Design and Implementation, pages 235-248, SIGPLAN Notices 27(7), 1992.

Lee et al.-2000 Lee H., Yoo C.: A Form Driven Object-Oriented Reverse Engineering Methodology. *Information Systems*, 25(3):235-259, 2000.

Leintz et al.-1980 Leintz B.P., Swanson E.F.: *Software maintenance Management*. Addison-Wesley, 1980.

Lien-1982 Lien Y.E.: On the Equivalence of Database Models. *Journal of the ACM*, 29(2), 1982.

Lopes et al.-2002 Lopes S., Petit J.-M., Toumani F.: Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems*, 27:1-19, 2002.

Markowitz et al.-1990 Markowitz V.M., Makowsky J.A.: Identifying Extended Entity-Relationship Object Structure in Relational Schemas. *IEEE transaction on software engineering*, 16(8):777-790, 1990.

Mills-1988 Mills, E.: Software Metrics. Technical report, SEI Curriculum Module SEI-CM-12-1.1, 1988.

Moonen - 2002 Moonen, L.: Exploring Software Systems.. PhD thesis, University of Amsterdam, The Netherlands. 2002.

Müller-1996 Müller H.: Understanding Software Systems Using Reverse Engineering Technologies Research and Practice. In *Proc. of 18th International Conference on Software Engineering*, Berlin, Germany, 1996.

Navathe-1980 Navathe S.B.: Schema Analysis for Database Restructuring. *ACM TODS*, 5(2), 1980.

Navathe et al.-1987 Navathe S.B., Awong A.M.: Abstracting Relational and Hierarchical Data with a Semantic Data Mode. In *Proc. of the 6th Internatial Conference on Entity-Relationship Approach (ER'87)*, pages 305-333, 1987.

Ottenstein et al.-1994 Ottenstein K.J., Ottenstein L.M.: The program dependence graph in a software development environment. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1994.

Petit et al.-1994 Petit J.-M., Kouloumdjian J., Bouliaut J.-F., Toumani F.: Using Queries to Improve Database Reverse Engineering. In *Proc. of the 13th int. Conf. on ER Approach (ER'94)*, Manchester, 1994. Springer-Verlag.

Petit-1996 Petit J-M. *Fondements pour un Processus Réaliste de Rétro-Conception de Bases de Données Relationnelles*. PhD thesis, University Lyon I, France, 1996.

Premerlani et al.-1993 Premerlani W.J., Blaha M.R.: An approach for Reverse Engineering of Relational Databases. In *Proc. of the Working Conf. on Reverse Engineering* (WCRE'93). IEEE Computer society Press, 1993.

Puerta et al.-2002 Puerta A., Eisenstein J.: XIML: A Common Representation for Interaction Data. In *Proc. of the IUI'02*, San Francisco, USA, 2002.

Quilici et al.-1997 Quilici A., Woods S., Zhang Y.: New Experiments With A Constraint-Based Approach To Program Plan Matching. In *Proc. of the ourth IEEE Working Conference on Reverse Engineering (WCRE'97)*, The Netherlands, 1997.

Ramanathan et al.-1996 Ramanathan S., Hodges J.: Reverse Engineering Relational Schemas to

Object-Oriented Schemas, techreport 960701, Department of Computer Science, Mississippi State University, 1996.

Rauh et al.-1995 Rauh O., Stickel E.: Standard Transformations for the Normalization of ER Schemata. In *Proc. of the CAiSE'95 conf.*, LNCS, Jyvaskyla, Finland, 1995. Springer-Verlag.

Robbins-2002 Robbins A.: *sed & awk Pocket Reference*. O'Reilly, 2002.

Roland et al.-2000 Roland D., Hainaut J.-L., Hick J.-M., Henrard J., Englebert V.: Database Engineering Processes with DB-MAIN. In *Proc. of the 8th European Conf. on Information Systems (ECIS2000)*, Vienna, Austria, 2000.

Rosenthal et al.-1988 Rosenthal A., Reiner D.: Theoretically Sound Transformation for Practical Database Design. In March, editor, *Proc. of the 6th International Conf. on Entity-relationship Approach (ER'98)*, 1988. North-Holland.

Rosenthal et al.-1994 Rosenthal A., Reiner D.: Tools and Transformations: Rigorous and Otherwise - for Practical Database Design. *ACM TODS*, 19(2), 1994.

Rugaber-1995 Rugaber S.: Program Comprehension. Technical report, Georgia Institute of Technology, 1995.
ftp://ftp.cc.gatech.edu/pub/groups/reverse/repository/encyc.ps.

Sellink et al.-2000 Sellink A., Verhoef C.: Scaffolding for Software Renovation. In *Proc. of the Conference on Software Maintenance and Reengineering (CSMR'2000)*, pages 161-172, Switzerland, 2000.

Signore et al.-1994 Signore O., Loffredo M., Gregori M., Cima M.: Using Procedural Patterns in Abstracting Relational Schemata. In *Proc. 3rd Workshop on Program Comprehension*, 1994.

Sneed-1991 Sneed H.: Economics of Software Re-engineering. *Software Maintenance and Practice*, 3:163-182, 1991.

Sneed-1995 Sneed H.: Planning the Reengineering of Legacy Systems. *IEEE Software*, 12(1):24-34, 1995.

Tan et al.-1997 Tan Hee., Ling T.: A method for the recovery of inclusion dependencies from data-intensive buisness programs. *Information and Software Technology*, 39:27-34, 1997.

Tangorra et al.-1995 Tangorra F., Chiarolla D.: A methodology for reverse engineering hierarchical databases. Information and Software Technology, 37(4):225-231, 1995.

Thiran et al.-2000 Thiran Ph., Chougrani A., Hainaut J.-L., Hick J.-M.: CASE Support for the Development of Federated Information Systems. In *Proc. of the 3rd International Workshop on Engineering Federated Information Systems (EFIS 2000)*, Dublin, 2000.

Tilley-1996 Tilley S.: Perspectives on Legacy System Reengineering. Technical report, Carnegie Mellon University, 1996.
http://www.sei.cmu.edu/reengineering/pubs/lsysree/lsysree.html.

Tilley-1998 Tilley S.: A reverse-engineering environment frameword. Technical report CMU/SEI-98-TR-005, Carnegie Mellon University, 1998.
http://www.sei.cmu.edu/publications/documents/98.reports/98tr005/
98tr005abstract.html.

Tilley-1998b Tilley S.: Coming attractions in program understanding II: Highlights of 1997 and opportunities in 1998. Technical Report CMU/SEI-98-TR-001, Carnegie Mellon University, 1998.

Tip-1994 Tip F.: A survey of program slicing techniques. Technical report CS-R9438, CWI, 1994.
ftp://ftp.cwi.nl/pub/CWIreports/AP/CS-R9438.ps.

van Deursen et al.-1998 van Deursen A., Kuipers T.: Rapid System Understanding: Two COBOL Case Studies. Technical Report SEN-R9805, CWI, The Netherlands, 1998.

Vermeer et al.-1995 Vermeer M, Apers P.: Reverse Engineering of Relational Database Applications. In *Proc. of the Object-Oriented and Entity-Relationship Modelling (OOER'95)*, 1995.

von Mayrhauser et al.-1993 von Mayrhauser A., Vans M.: From Program Comprehension to Tool Requirements for an Industrial Environment. In *Proc. of Second Workshop on Program Comprehension*, 1993.

von Mayrhauser et al.-1994 von Mayrhauser A., Vans A. M.: Program Understanding -- A Survey. Technical Report CS-94-120, Colorado State University, Computer Science Department, 1994.
http://www.cs.colostate.edu/~ftppub/TechReports/1994/tr-120.pdf.

Weiser-1984 Weiser M.: Program Slicing. *IEEE TSE*, 10(4):352-357, 1984.

Wilde-1990 Wilde N.: Understanding program dependencies. Technical report CM-26, 1990 http://www.sei.cmu.edu/publications/documents/cms/cm.026.html.

Winans et al. - 1990 Winans J., Davis K.: Software Reverse Engineering from a Currently Existing IMS Database to an Entity-Relationship Model, In *Proceedings of the 9th International Conference on Entity-Relationship Approach (ER'90)*, pages 345-360, Switzerland, 1990.

Winter - 2002 Winter A.: GXL - Overview and Current Status. In *Proc of the Int. Workshop on Graph-Based Tools (GraBaTs)*, Barcelona, Spain, 2002.

XMI - 2002 OMG XML Metadata Interchange (XMI) Specification, 2002.

Young-1996 Young P.: Program Comprehension. Technical report, Center for Software Maintenance, 1996.
http://www.dur.ac.uk/~dcs3py/pages/work/Documents/

FUNDP
Institut d'Informatique
Rue Grandgagnage, 21
B-5000 Namur
Belgique

PROGRAM UNDERSTANDING IN DATABASE REVERSE ENGINEERING
ANNEX

Jean HENRARD

Thesis submitted for the degree of Doctor of Science
(Computer Science Option)

Jury : Professor Jean Fichefet, Institut d'informatique, FUNDP (President)
Professor Jean-Luc Hainaut, Institut d'informatique, FUNDP (Supervisor)
Doctor Rainer Koschke, Universität Stuttgart, Germany
Doctor Jean-Marc Petit, Université Blaise Pascal, Clermond-Ferrand, France
Professor Jean-Marie Jacquet, Institut d'informatique, FUNDP

August 2003

ANNEX A DBRE tools user manual

This annex presents the user manual of the DB-MAIN specific DBRE tools

A.1. Pattern definition language

The pattern definition language (PDL) is used to define the patterns to be used in the search tool (**Assist - Text analysis - Search**) (A.2) by a procedure triggered by a pattern (**Assist - Text analysis - Execute**) (A.3) and by the variable dependency graph (**Assist - Text analysis - Dependency**) (A.4). This section describes the PDL syntax and how to load a PDL file into DB-MAIN.

A.1.1. The syntax

The PDL syntax is given as a BNF grammar. The non terminal element are noted by <...> and the reserved symbols of the language are in bold.

```
<pattern>:
       <pattern_name>::= <segment>*;

<segment>:
       <terminal_seg>
     | <pattern_name>
     | <variable>
     | <range>
     | <optional_seg>
     | <repeat_seg>
     | <group_seg>
     | <choice_seg>
     | <regular_expr>

<variable>:
       @<pattern_name>
```

The '@' symbol indicates that the segment is a variable. If a variable appears two times in a segment, then both occurrences have the same value. When a pattern is found, the value of the variables can be known. A variable can not appear in a repetitive structure.

```
<range>:
        range(c1-c2)
```
Is any character between c1 and c2. c1 and c2 are two characters.

```
<optional_seg>:
        [<segment>]
```
Optional segment

```
<repeat_seg>:
         <segment>*
```
Repetitive segment

```
<group_seg>:
        (<segment>*)

<choice_seg>:
        {<segment> | ... | <segment>}
```
Any of the segment.

```
<regular_exp>:
        /g"<a regular expression>"

<terminal_seg>:
        "a string"
```
/t = tabulation; /r/n = new line

```
<pattern_name>:
        [A-Za-z0-9][A-Za-z0-9]0-29
```

The characters that form regular expressions (<a regular expression>) are:

- `.` Matches any single character.
- `*` Matches 0 or more copies of the preceding expression.
- `+` Matches 1 or more copies of the preceding expression.
- `[...]` Matches any character within the brackets, e.g. `[0,1,2]` means `0` or `1` or `2`.
- `[x-y]` Is a notation for a character range, e.g., `[0-4]` means `[0,1,2,3,4]`.
- `?` Matches 0 or 1 occurrence of the previous expression.
- `"..."` Matches exactly the content enclosed between quotes.
- `/t` Denotes the tabular.
- `/r/n` Denotes the newline characters.

A forward reference is not allowed in a pattern definition. That means that if a pattern is used in the definition of another pattern, that pattern must be defined before.

A.1.2. Examples

```
- ::=   /g"[/r/n/t ]+";
```

The name of the pattern is `-`, its definition is `/g"[/r/n/t ]+"`. This pattern is a regular expression (`/g"..."`). The regular expression `[...]` matches any characters within the brackets and the `+` matches one or more occurrences of the preceding expression. This pattern matches at least one "space" (space, new line or tab).

```
~ ::= /g"[/r/n/t ]*";
```

Almost the same as the previous one, except `*` matches zero or more occurrences of the preceding expression.

```
var ::= /g"[a-zA-Z][-a-zA-Z0-9]*";
```

A COBOL variable.

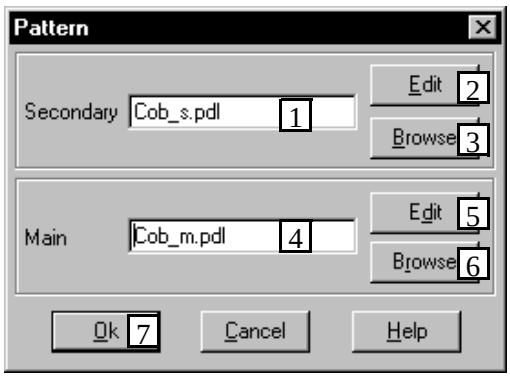
```
var_1 ::= var;
var_2 ::= var;
```

Those two patterns are just two different names to the pattern `var`.

```
move ::= "MOVE" - @var_1 - "TO" - @var_2 ;
```

This pattern matches the COBOL move instruction. It matches the characters `"MOVE"`, followed by the pattern `-` (a mandatory space), followed by the pattern `var_1` that is assigned to a variable named `var_1` (because of the `@`), followed by the pattern `-`, followed by the characters `"TO"`, followed by the pattern `-`, followed by the pattern `var_2` that is assigned to the variable `var_2`.

A.1.3. In DB-MAIN



1. The secondary pattern file.
2. Edit the secondary pattern file.
3. Changes the name of the secondary pattern file.
4. The main pattern file.
5. Edits the main pattern file.
6. Changes the name of the main pattern file.
7. Compiles the pattern files.

FIGURE 171. The load/edit pattern dialog box.

The patterns are stored into two text files, main and secondary. The secondary patterns file contains the definitions of patterns that are the basic patterns used by other patterns. The secondary patterns file contains for example the definition of the spaces (mandatory or not), of the variables in the target language. On the other side, the main patterns file contains, for example, the definition of the assignment, of the comparison.

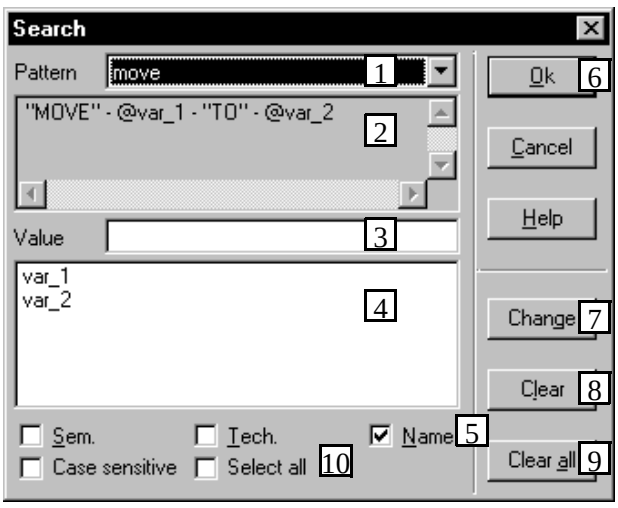
To specify the patterns files to load, use the **Assist - Text analysis - Load pattern** command. The **Load / Edit Pattern** dialog appears (figure 171). The top part of the window contains the name of the secondary file. The button **Edit** (2) is used to edit the secondary file and the **Browse** (3) button is used to browse the disks to find the secondary patterns file. The middle part of the window contains the name of the main patterns file. The button **Edit** (5) is used to edit the main file and the **Browse** (6) button is used to browse the disks to find the main patterns file.

When the **Ok** button is clicked, the patterns are compiled and become the patterns used by different text analysis tools as search and variable dependency graph. If an error occurs during the compilation, an error message is displayed and there is no pattern available in the text analysis tools.

A.2. Search for text pattern

The command **Assist - Text analysis - Search** is used to search for a pattern in a text file or in the descriptions or the names of the objects of a schema. The variables of the pattern can be instantiated before the search. When a pattern matches a string, it is possible to see the value assigned to the variables.

A.2.1. Search for a pattern



1. The list of the patterns.
2. The definition of the selected pattern.
3. Text edit used to change the value of a variable.
4. The list of the variables.
5. In a schema, search in the semantic and/or technical descriptions and/or the object's name.
6. Start the search.
7. Change the value of the current variable to the value displayed in (3).
8. Clear the current variable.
9. Clear all the variables.
10. If checked, the search is case sensitive.
11. If checked, all the lines or the objects that contain the pattern are selected.

FIGURE 172. The Search dialog box.

Assist/Text analysis/Search (<ctrl-F>) is used to search for a pattern in the Search dialog box (figure 172). The combo box **Pattern** (1) contains the list of all the defined patterns. The definition of the selected pattern appears in the text below (2) and its variables appear in the bottom list box (4), with their values if they are instantiated. The first pattern of the list (`user def`) is a special one that has no definition, and it is up to the user to write it in the text below, the usual PDL syntax must

be used. This pattern is used to look for a "one shot" pattern, that is not saved into the pattern files. If case sensitive is checked (10), then the search is case sensitive.

If a variable is instantiated before the search, the variable is replaced by its value. Otherwise the variable is replaced by its definition. To instantiate a variable, selected it into the list of variables, type its value into the **Value** text box (3) and then click on the **Change** button (7). Its new value appears into the list of variables. To clear the value of the selected variable (to uninstantiate it), click on the **Clear** button (8). To clear the value of all the variables (to uninstanciate all to them), click on the **Clear all** button (9).

If the search take place into a schema, three check boxes appear in the bottom of the window. Check one or all of them to specify if the search must take place into the semantic or/and technical descriptions or/and in the name of the objects.

If the **Select all** button is checked, all the lines (if the search take place into a text) or all the objects (if the search take place into a schema) that contains the pattern are selected. Otherwise only the next line or the next object that matches the pattern is selected.

Click on the **OK** button to start the search.

If the search take place into a text file, it starts at the line that follows the current line, if there is no current line, it starts at the first one. And it goes from one line to the next until the pattern match.

If it take place into a textual view of a schema, its start at the object that follow the current one.

If it take place into a graphical view of a schema, the order of the search is unpredictable. If no object is selected, the search take place in all the schema.

A.2.2. Search next

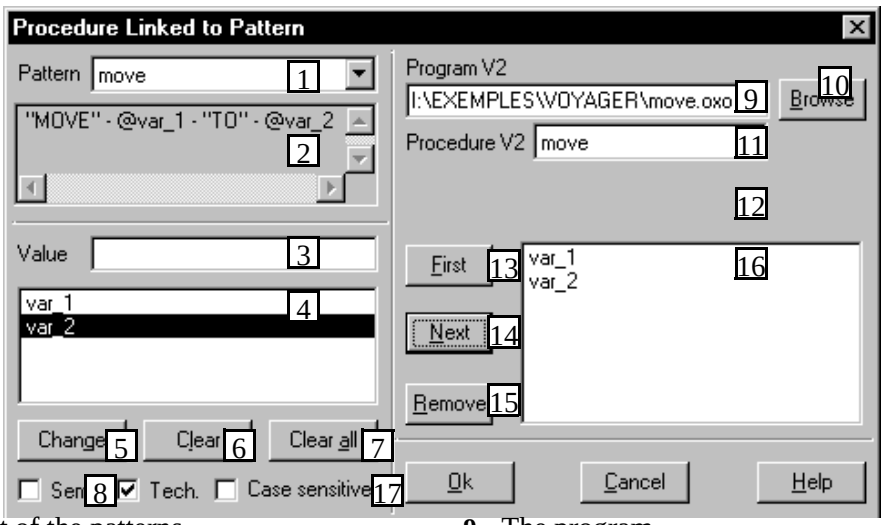
To search for the next occurrence of the pattern use the command **Assist - Text analysis - Search next** or the <F3 > key.

A.3. Procedure triggered by a pattern

It is possible to search for a pattern into a text or into the description of a schema and each time the pattern matches a procedure is executed. The variables of the pattern are the parameters of the procedure. The pattern is search until the end of the text or of the schema.

To execute a procedure when a pattern is found can be useful to automate tasks. For example, if the views into a SQL-DDL represent subtype of table, it is possible to search for the entity types that represent views into the schema (their definition are in the technical description of the entity types) and for each of them create the is-a relation to connect them to their super-type (the table).

A.3.1. Usage



1. The list of the patterns.
2. The definition of the selected pattern.
3. The value of the current variable, used to edit its value.
4. The list of the variables.
5. Changes the value of the current variable (4).
6. Clears the current variable (4).
7. Clears all the variables.
8. In a schema, search in the semantic and/or technical descriptions.
9. The program.
10. Changes the program.
11. The procedure to be executed.
12. The description of the procedure.
13. Adds the current variable (4) as the first parameter of the procedure (16).
14. Adds the current variable (4) as the next parameter of the procedure (16).
15. Removes the current parameter (16).
16. List of the parameters of the procedure.
17. If checked, the search is case sensitive.

FIGURE 173. The procedure triggered by a pattern dialog box.

The command **Assist/Text analysis/Execute** allows executing a *Voyager2* procedure each time a pattern is found.

The left side of the Procedure linked to Pattern dialog (figure 173) is the same as the Search dialog box. Except that, if it is call from a text file there are three pseudo-variables (`file name`, `line num`, `pattern`) and if it is call from a schema there is a pseudo-variable (`pattern`). `File name` is instantiated with the name of the file in which the search takes place, `line num` is the number of the line in which start the pattern (the pattern may be on several lines) and `pattern` is the instantiation of the pattern.

The right side of the dialog box contains the procedure to be executed each time the pattern on the left side is found. The **Program V2** text box (9) contains the name of the program (`oxo` file), the **Browse** button (10) can be used to find the program. When a program is selected, the **Procedure V2** combo box (11) contains the list of procedures exported by the program. Select the procedure to be executed.

The list box below the name of the procedure (16) must be filled by the variables of the pattern that are used as parameter of the procedure. To fill this list, select a variable in the list of the pattern's

variables (4) and the click on the **First** (13) or **Next** (14) buttons. To remove a variable from the list (16), select it and click on the **Remove** button (15).

Click on the **OK** button, then the search start at the selected line (if the search take place into a text file) or at the current object, until the end. Each time the pattern match, the procedure is executed.

A.3.2. Example (1)

This example shows how to generate a report of all the assignment instructions found into a COBOL program. For each move instruction, the line number of the instruction and the two variables (the origin and the target of the assignment) are printed.

The patterns used to search into the source text are the following:

```
-         ::= /g"[/n/t/r ]+";
~         ::= /g"[/n/t/r ]*";
var       ::= /g"[a-zA-Z][-a-zA-Z0-9]*";
var_1     ::= var;
var_2     ::= var;
move      ::= "MOVE" - @var_1 - "TO" - @var_2 ;
```

The `move` procedure is called to display the report, the procedure is declared `export`, because it must be call from outside the voyager program.

```
export procedure move(string: var_1,
string : var_2,
         string : file_name, string : line)
{
     SetPrintList("","","");
     print ([file_name, ",", line, " : ",
            var_2, "-->", var_1, "\n"]);
}
```

A.3.3. Example (2)

In this example, SQL views represent sub-types of a table. The views are defined as follow

```
create view (.....)
as select (.......)
from <table>
where <column> = <string>;
```

The SQL extractor extracts views as entity types and puts the declaration of the views into the technical description of the entity type. The tables are extracted as entity types.

The purpose of the example is to create is-a relations between the super types (the tables) and their sub-type (the views). To know to which table a view must be connected, the declaration of the view (stored into the technical description) is search for the follow code fragment:

```
from <table> where <column> = <string>
```

When such a fragment is found, an is-a relation can be created between the view (the entity type that contains the code fragment in its technical description) and the table (the entity type of name <name>)..

The patterns of figure 174 are used to search into the technical description are the following.

```
- ::= /g"[/n/t/r ]+";
string ::= /g"'.*'";
name ::= /g"[a-zA-ZO-9_]+";
table ::= name;
column ::= name;
from ::= "from" - @table - "where" - column - "=" - string;
```

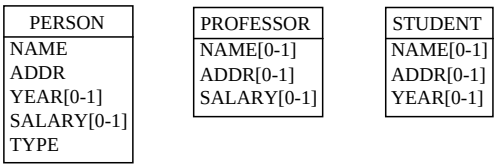
FIGURE 174. Declaration of the pattern `from`.

The Voyager2 function, create_is_a, is called to create the is-a relation. It has one argument, the name of the super-type (the table). It creates the is-a between the super-type and the current entity type (the view).

```
export procedure create_is-a(string: table)
/* creates a is-a relation between the entity type of name
     'table' and the current entity type*/
  data_object : d_obj;
  schema : sch;
  entity_type : sub_ent;
  entity_type : super_ent;
{
  SetPrintList("","","");

  sch := GetCurrentSchema();
/* get the current schema*/
  if IsVoid(sch) then {
/* if there no current schema return an error */
    print("No Schema !\n");
    return;
  }

  go :=GetCurrentObject();
/* get the current object */
  if IsVoid(go)
  then {
/* if there is no current object, return an error */
    print("No current object !\n");
    return;
  }
  if (GetType(go) <> ENTITY_TYPE)
  then {
/* if the current object is not an entity type, return an
     error */
    print("The current object is not a entity type !\n");
    return;
  }
  sub_ent := go;

/* 'sup_ent' is the entity type of name 'table' */
  sup_ent := GetFirst(DATA_OBJECT[d_obj]{@SCH_DATA:[sch]
      with ((GetType(d_obj) = ENTITY_TYPE)
            and (d_obj.name = table))});

/* 'l_clu' is the list of cluster connected to the super
     type*/
  l_clu := CLUSTER[clu]{@ENTITY_CLU:[sup_ent]};

  if(Length(l_clu) = 1) then
  {
```

```
      /* if the super type has a cluster, use it */
    clu := GetFirst(l_clu);
  }
  else
  {
/* if the super type has no cluster, create it */
    clu := create(CLUSTER, name : sup_ent.name, total : 0,
            disjoint : 0, @ENTITY_CLU : sup_ent);
  }

/* connect the sub-type to the cluster */
  sub_t := create(SUB_TYPE, @CLU_SUB : clu,
                  @ENTITY_SUB : sub_ent);
}
```

The usage of this pattern and of the `create_is_a` procedure will be illustrate on an example. The procedure triggered a pattern tool is used on the raw physical schema of figure 175. To ease the understanding of the schema the view declaration code have been represented as textual annotation.




```
create view PROFESSOR (NAME, ADDR, SALARY) as
  select NAME, ADDR, SALARY from PERSON
    where TYPE = 'P';
create view STUDENT (NAME, ADDR, YEAR) as
  select NAME, ADDR, YEAR from PERSON
    where TYPE = 'S';
```

FIGURE 175. The raw physical schema.

To create the is-a relation, nothing is selected into the schema and the command **Assist - Text analysis - Execute** is executed. from is selected as the pattern, `create_is_a.oxo` as the program, `create_is_a` as the procedure and the procedure has only one parameter (`table`).

When the **Ok** button is clicked, the is-a relation is created to produce the refine schema of figure 176

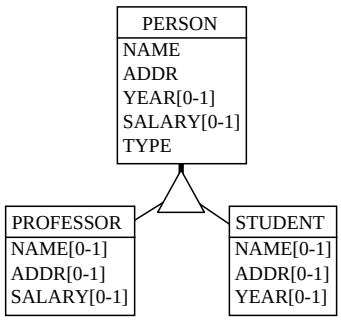


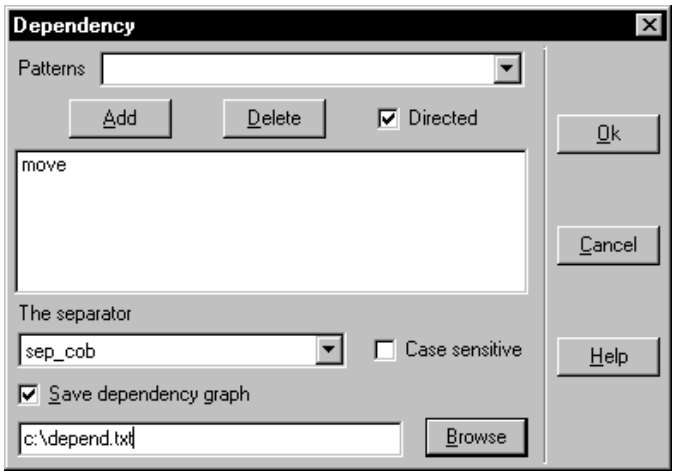
FIGURE 176. The Schema with the is-a relation created.

A.4. Dependency graph

The are three steps to use the dependency graph in DB-MAIN:

- Computes the dependency graph itself.
- Changes the settings of the graph visualization.
- Visualizes the graph.

A.4.1. Computes the dependency graph



1. List of patterns used to compute the dependency graph.
2. List of available patterns.
3. Adds the current pattern (2) to the list of patterns used to compute the dependency graph (1).
4. Remove the selected pattern from the list.
5. If checked, the next pattern to be added is oriented (from `var_1` to `var_2`).
6. Name of the file in which the dependency graph will be saved.
7. Opens the standard file dialog box to change the name of the file.
8. If checked, saves the dependency graph into a file.
9. If checked, the pattern is case sensitive.
10. The pattern use to find the beginning and the end of the variables.

FIGURE 177. The variable dependency graph dialog box.

The dependency graph can be computed by the command **Assist - Text analysis - Dependency** (figure 177). The relation between two variables is given by a pattern, which contains two PDL variables (`var_1` and `var_2`). The list box (1) contains the list of patterns used to compute the dependency graph. To add a pattern to this list, select it into the combo box (2), check the check-box **oriented** (5) the pattern must be oriented and then click on the **Add** button (3). To remove a pattern from the list, select it and click on the **Delete** button (4).

All the patterns of the list box must have two variables named `var_1` and `var_2`. The lists of variables contained into `var_1` and `var_2` are computed using the pattern "separator" (see below) to separate the variables. If the arcs are oriented, they go from all the variables of `var_1` to all the variables of `var_2`.

The separator (10) is a pattern that marks the beginning and the end of a variable name (the separator). This pattern can match only one character string.

If **Save dependency graph** (8) is checked, the dependency graph is saved into the given file (6). The file contains two version of the dependency graph: one that contains only the relation found

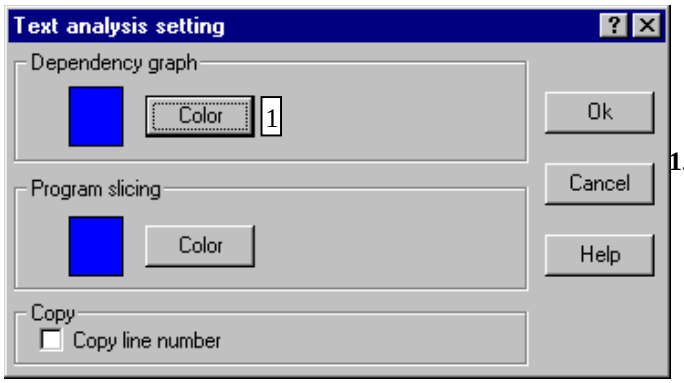
using the patterns and the one that is the transitive closure of the first one. The graphs are stored in a textual format

```
 <variable> : <list of the variables directly reachable from the
variable>
```

The two graphs are separated by a line "****".

Click on **OK** to compute the dependency graph. It can take some time, depending on the size of the text and of the number of patterns in the list.

A.4.2. Change the settings



1. Change the color in which the variables bellowing to the dependency graph are colored.

FIGURE 178. The text analysis setting dialog box.

The text analysis setting dialogue can be reach by the command **Assist - Text analysis - Setting** (figure 178). Only the top part of the dialog is related to the dependency graph configuration.

The **Color** (1) button is used to change the color in which the dependency graph will be colored (see later). The color used is displayed into the square on the left.

A.4.3. Visualization of the dependency graph

```
219  LECTURE-DETAIL.
220    DISPLAY "CODE DU PRODUIT (0 = FIN) : ".
221    ACCEPT CODE-PROD.
222    IF CODE-PROD = "0"
223      MOVE 0 TO FIN-FICHIER
224      MOVE 0 TO REF-STOCK-DE(IND-DET)
225    ELSE
226      PERFORM LECTURE-CODE-PROD.
227
228  LECTURE-CODE-PROD.
229    MOVE 1 TO EXIST-PROD.
230    MOVE CODE-PROD TO STK-CODE.
231    READ STOCK INVALID KEY
```

When the dependency graph is computed. Click with the mouse's right button on a variable into the text file and if this variable belong to the dependency graph then all the variables backward or forward reachable (directly or indirectly) to this variable are colored, everywhere in the text file.

Select a line with the left button of the mouse and then press the <Tab> key. The next line that contains a colored variable will be displayed in the middle of the screen.

The pattern select into the **The separator** combo box is use at this level to find the beginning and the end of the variable on which you click.

A.4.4. Configuration

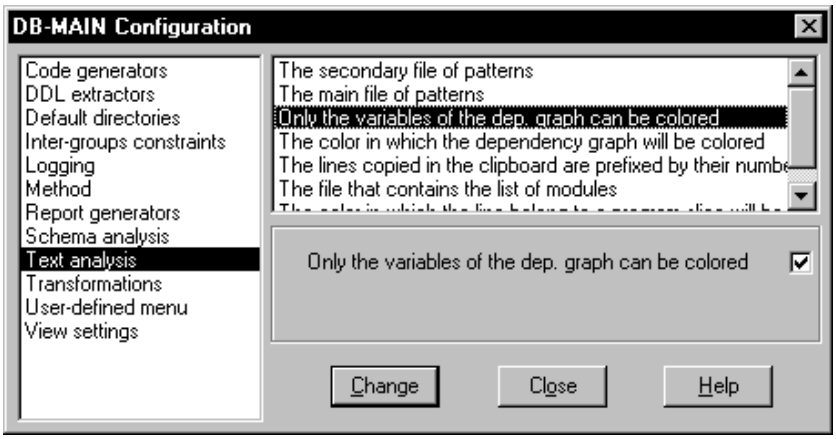


FIGURE 179. Configure the dependency graph visualization.

The normal behavior of the dependency graph, is that only the variables belonging to the dependency graph are colored. It can be useful to color all the occurrences of a variable even if it does not appear in the dependency graph.

To change the behavior of the dependency graph tool, select the command **File - Configuration**. The configuration dialogue appears. Select the *Text analysis* item into the left list box. Then the "*Only the variables of the dependency graph can be colored*" is displayed, selected it. If you want the tool colors only the variable belong to the dependency graph, unchecked the check box and click on the **Change** button. If you want that the tool colored the variables even if they do not appear in the dependency graph check the check box and click on the **Change** button.

When variables that are not in the dependency graph are colored, the tool can be used to color word that are not variable. For example, if you click on the name of a procedure, each call to the procedure and its declaration are colored.

A.4.5. Tips

1. If the combo box containing the available patterns is empty, it means that there is no pattern loaded or there is a syntax error in one of the pattern. See **Assist - Text analysis - Load pattern**.
2. To add a new pattern or to modify an existing one, use the **Assist - text analysis - Load pattern**.
3. To check if the patterns used to compute the dependency graph are correct, use the command **Assist - Text analysis - Search** to check that they are correct and that they match with the expected instructions.

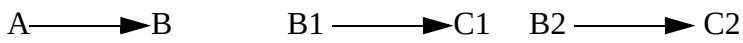
A.4.6. Remarks

The computation of the variable dependency graph is a syntactical process (it only uses the patterns). The usage of the patterns to construct the dependency graph make the tool very flexible and easily customizable to almost any language.

But the drawback is that the dependency graph is not award of the variables structure or of the program's control flow. This can lead to an incomplete graph as in the following example:

```
01 A pic x(10).
01 B.                          move A to B.
  02 B1 pic x(5).              ...
  02 B2 pic x(5).              move B1 to C1.
01 C.                          ...
  02 C1 pic x(5).              move B2 to C2.
  02 C2 pic x(5).
```

The dependency graph is the following:

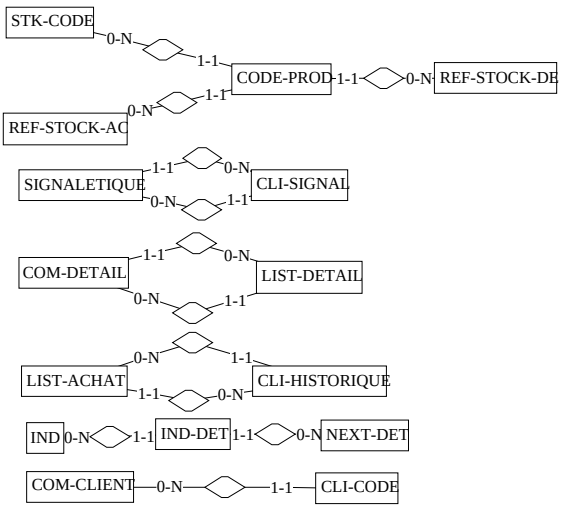


and the relation between `A` and `C` is not present, because the graph is not awarded that `B1` and `B2` are the two component of `B`.

A.4.7. Dependency graph visualization

A.4.7.1. Graph drawing

```
IND-DET : IND NEXT-DET
NEXT-DET :
SIGNALETIQUE : CLI-SIGNAL
CLI-SIGNAL : SIGNALETIQUE
CLI-CODE : COM-CLIENT
COM-CLIENT :
CLI-HISTORIQUE : LIST-ACHAT
LIST-ACHAT : CLI-HISTORIQUE
LIST-DETAIL : COM-DETAIL
COM-DETAIL : LIST-DETAIL
CODE-PROD : REF-STOCK-AC
   REF-STOCK-DE STK-CODE
STK-CODE :
REF-STOCK-DE :
REF-STOCK-AC :
IND :
```



The `depend.oxo` program can be used to visualize the dependency graph. It displays it as an entity/relationship schema, where the entity types represent variables and the relationship types represent the arcs.

To use it, create an empty schema (as the current window) and execute `depend.oxo`. Give as parameter the file generated by the computation of the dependency graph.

A.4.7.2. Mark graph

The `mark_dp.oxo` program is used to mark the entity types in the dependency graph that represent entity types or attributes of the data schema. Before executing `mark_dp.oxo`, make sure that de dependency graph is in the current window. The program asks you the schema that contains the data schema.

A.5. Program slicing

A.5.1. Use of program slicing

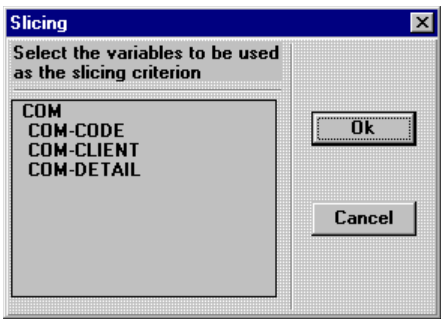


FIGURE 180. Dialog box to select the variables with respect to the program slice must be calculated.

Select the line with respect to which you want to compute the program slice, then use the command **Assist - Text analysis - Program slicing**. Then a dialog containing the variables referenced into the selected instruction is displayed. Select the variable(s) for which the slice must be calculated. The slice is colored, if an instruction is on several lines, only the first one is colored.

The first time you use the program slicing tool on a COBOL source code it can take some time (several minutes for big programs) because it must parse the program and create the system dependency graph. The next time a program slice is computed for this source code it goes faster, because the computation of the program slice consists only into the traversing of the system dependency graph.

During the parsing of the program, syntax error (syntax that are not understood) are displayed, but the parsing continue. The sentence (until the next ".") where the syntax error occurs is not represented into the system dependency graph, so is ignored during the computation of a slice. So the slice can be incomplete because of the "syntax error".

A.5.2. Call graph

One of the side effect of the program slicing tool is that it creates a processing schema that represent the program call graph, named `file_name/prg`. Each procedure is represented by a processing unit named as the procedure (section or paragraph) and the perform instruction by a call relation.

The command **Assist / Text analysis / Goto** can also be used to go from the call graph to the text and from the text (section or paragraph) to the call graph.

A.5.3. The command line program slicing

A.5.3.1. Description

We have derived from the DB-MAIN program slicing tool a command line program slicing tool. This tool can be very useful to compute slices for very big programs (it can run on a Unix workstation) or to compute slice of many different programs (batch processing). It computes slices of the `<cobol_file>` COBOL program, if no `<cobol_file>` is given, then it asks for one.

A.5.3.2. Synopsis

```
slicing   [-d] [-f] [-s start -c condition [-e node_type] [-v]]
          [-a action] [-g sdg_file] [-p parse_file] -F <command_file>
          [-o  output]  [-i]  [-P  <beg>  <end>]  [-b  <db_structure>]
          [cobol_file]
```

A.5.3.3. Option

-d	Computes the slice using only data dependence arcs. By default, it uses also the control dependence graph.
-i	Doesn't include the "invalid key" branch into the SDG.
-f	Computes the forward slice, the default value is to compute the backward slice.
-g <sdg_file>	Saves the SDG corresponding to the COBOL program into <sdg_file>.
-p <parse_file>	Saves the parsing tree corresponding to the COBOL program into <parse_file>.
-o <output>	Displays all the messages into the file <output>, by default they are displayed to the standard output.
-v	Computes which part of the variables defined in the condition node influence which part of the variable referenced in the start node. Need the options -s, -c
-s <start>	*<start>* is the type of nodes with respect which the program slice will be computed. The possible values of *<start>* are `read`, `write`, `proc`, `normal`, `test`, `loop`, `goto`, `perform` or `line:<l_1>...<l_n>` with `l_i` an integer (line number). If no *<start>* is given, the user is asked for a line number.
-c <condition>	If this option is used, only the lines of the slice that are of the given type will be displayed. By default, all the lines are displayed. The valid value of *<condition>* are `read`, `write`, `proc`, `normal`, `test`, `loop`, `goto`, `perform`.
-a <display>	Defines how the result will be displayed. The valid *<display>* are `num` (only the number of the lines will be displayed), `line` (the complete lines will be displayed), `num_line` (as num and line) and `var` (only the variables referenced at the instruction will be displayed).
-e <node_type>	Stops the SDG traversing when a node of type *<node_type>* is reach. The valid value of <node_type> are `read`, `write`, `proc`, `normal`, `test`, `loop`, `goto`, `perform`.

-P <begin> <end> Marks only the lines of the slices that are in the path between `<begin>` and `<end>`.

-b <db_structure> A file describing the declaration of the database used in the "`SCHEMA SECTION`".

-F <command_file> Each line of the file *<command_file>* describe a slice (or a SDG traversal) to compute. Each line contains some valid options, with the same syntax as the option of `slicing`. The valid options are: `[-f] [-s start -c condition [-e node_type] [-v]] [-a action] [-P <beg> <end>]`. For example, if *<command_file>* contains two lines, it is equivalent to two executions of `slicing` with the options of each lines. Except that if we use *<commande_file>*, the SDG is only computed one and if we use two `slicing` command the SDG is computed twice. For big programs the computation of the SDG can take several hours on a powerful workstation.

A.6. Creating schema

Many DBRE projects require to extract information from source code, text and represent them as a schema into DB-MAIN. DB-MAIN offers some built-in extractors (SQL, COBOL, etc.) and some specific extractors (XML/DTD, RPG, etc.) have been developed in *Voyager2*. Those extractors are not sufficient to solve all the specific needs of all the projects. On the other hand, it can be to expensive to write, in *Voyager2*, a specific extractor for a given project. To solve this problem there exist two generic schema extractors. These extractors do not take as input a source code, but an intermediate text file that describe the schema. This intermediate text file can be easily generate from the source code by some scripting language such as grep, awk, perl, etc.

The two programs can be used to create a processing schema for the first and a data schema for the second.

A.6.1. Processing schema

The `graph_tr.oxo` program creates a processing schema from an input file. This program can create processing units, data objects, call relations, decomposition relations and in-out relations.

A.6.1.1. The input file

To create a graph the `graph_tr.oxo` program needs a file that contains lines with the following format:

```
<line_type>;<param_1>;...;<param_n>
```

where `<line_type>` = "PROC", "VAR", "CALL", "IN", "OUT", "I-O" or "DECOMP"

If `<line_type>` == "PROC" ou "VAR"

```
<param_1> = <node_type>
<param_2> = <node_name>
```

```
<param_3> = <second_node_name>
<param_4> = <node_type>
<param_5> = <description>
```

Creates a processing unit (<line_type> = "PROC") or a data object (<line_type> = "VAR") of name <node_name> and the dynamic property S_name and Type take respectively the value of <second_node_name> and <node_type>. The node name and its dynamic property S_name are both identifiers. Description is added to the technical description of the processing unit or of the data object. If a node with the same name or the same dynamic property S_name exists, the new node is not created.

If <line_type> == "CALL", "DECOMP", "IN", "OUT" or "I-O"

```
<param_1> = <type>
<param_2> = S or P
<param_3> = <name_1>
<param_4> = S ou P
<param_5> = <name_2>
```

If <line_type> is "CALL" or "DECOMP", creates a relation call or decomposition. The relation connect the processing unit <name_1> to the processing unit <name_2>. <name_1> (<name_2>) represents the name of a processing unit, if <param_3> (<param_5>) equal P and it represents the S_name, if <param_3> (<param_5>) equal S. <type> is store in the dynamic property Type. If the processing unit <name_1> or <name_2> does not exist the relation is not created.

If <line_type> is "IN" or "OUT" or "I-0", creates a relation in-out. The relation connects the processing unit <name_1> to the variable <name_2>. <name_1> (<name_2>) represents the name of a processing unit (or data object), if <param_3> (<param_5>) equal P and it represents the S_name, if <param_3> (<param_5>) equal S. If the processing unit <name_1> or the variable <name_2> does not exist the relation is not created. If there is no relation between <name_1> and <name_2>, it is created of type in input (<line_type> = "IN"), output (<line_type> = "OUT") or update (<line_type> = "I-O"). Else the relation is updated to add the new type. I.e. if it was of type input and the <line_type> is "OUT" or "I-O", then the type of the relation is change to update; if it was of type output and the <line_type> is "IN" or "I-O", then the type of the relation is change to update; otherwise the type of the relation is not modified.

A.6.1.2. The creation of the graph

The graph_tr.oxo program adds the processing units, variables and relations described in a file to the current processing schema.

To create a new graph:

- Open (or create) a processing schema.
- Execute the graph_tr.oxo program (**File - Execute voyager**) and give it a file of the format described in the previous section. The file can not contain forward references.

The graph is created, but the processing units and variables are not correctly positioned. For big schema, it can be very painful to position each object, on the other hand the DB-MAIN build-in tools (auto draw) are not very useful because they are for ordinary processing schema and our graph

has other graphical properties such as it is not any graph but it can be a tree,... This is why we have written a series of small *voyager2* programs that help the analyst to manipulate the graph.

A.6.1.3. Data schema coloring

The `color_tr.oxo` program colors processing units depending of their dynamic property 'Type': asks the user the value of the dynamic property `Type` and its associates color (to be chosen into a list).

A.6.1.4. Sort graph by level

The `sort_n_tr.oxo` program sorts by level the current schema according to the call relation.

A.6.1.5. Center entity types

The `g_center_tr.oxo` program centers (horizontally) the marked (in **Mark1** mark plan) processing units with respect to the processing units that can be reach through call relation (depending of the chosen option).

A.6.1.6. Mark objects with the schema analysis

It is possible to mark or select objects using the schema analysis assistant (**Assist/Text analysis**). For example, we can select all the processing units that are not connected to other ones, etc.

A.6.1.7. Mark processing units reachable following call relations

The `g_slice_tr.oxo` program travels through the schema from the selected processing units following the call relations (the direction depends of the chosen option) and marks all the processing units crossed. It is useful to extract a branch of a call graph.

A.6.2. Data schema

The graph.oxo program creates a data schema from an input file. This program can create entity types, relationship types (with roles), attributes.

A.6.2.1. The input file

To create a graph the `graph.oxo` program needs a file that contains lines with the following format:

```
<line_type>;<param_1>;...;<param_n>
```

where `<line_type>` = "ET", "RT"

If `<line_type>` == "ET"

```
<param_1> = <node_type>
<param_2> = <node_name>
<param_3> = <second_node_name>
```

Creates an entity type of name <node_name> and the dynamic property S_name takes the value of <second_node_name>. The node name and its dynamic property S_name are both identifiers. If a node with the same name or the same dynamic property S_name exists, the new node is not created.

If <line_type> == "RT"

```
<param_1> = <node_name>
<param_2> = S or P
<param_3> = <name_1>
<param_4> = S or P
<param_5> = <name_2>
```

Creates the relationship-type of name <node_name> (add a suffix to have a unique name). Connects the relationship-type to the entity type <name_1> with a 1-1 role and to the entity type <name_2> with a 0-N role. <name_1> and <name_2> are the name of the entity type if they are preceded by P and the value of the dynamic property <S_name> if they are preceded by S.

A.6.2.2. The creation of the graph

The graph.oxo program adds the processing units, variables and relations described in a file to the current processing schema.

To create a new graph:

- Open (or create) a data schema.
- Execute the graph.oxo program (**File - Execute voyager**) and give it a file of the format described in the previous section. The file can not contain forward references.

A.7. Search a schema for referential constraints

A.7.1. About referential constraints assistant

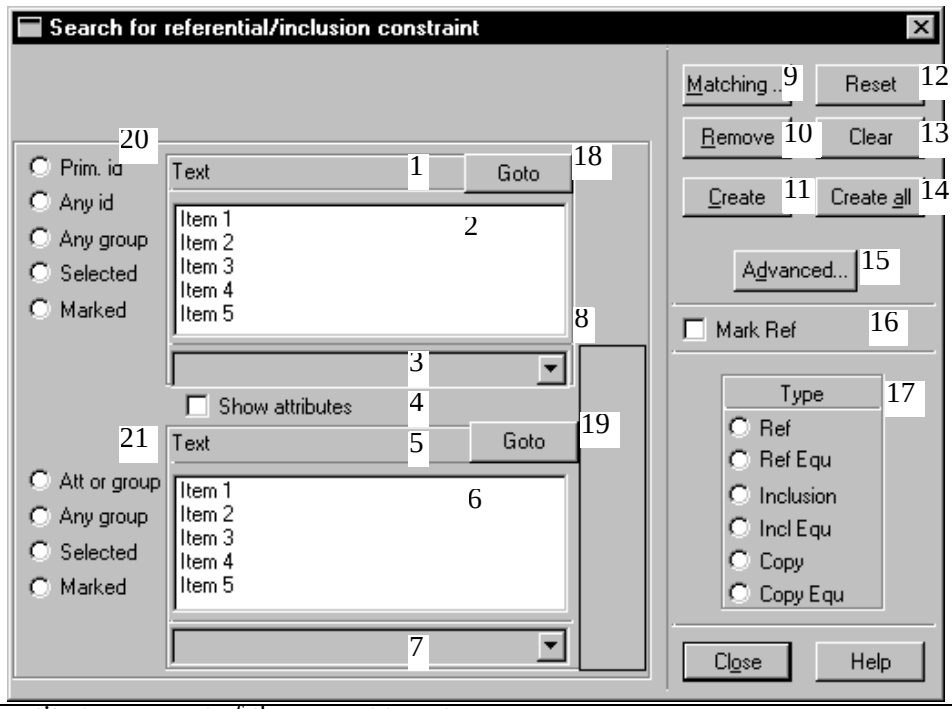
The referential constraints assistant proposes some popular heuristics to find and to create referential constraints.

The different heuristics are composed of two kinds of rules. The first one are the rules to find the target and origin candidate of the referential constraints. For example, the target of the referential constraints must be a primary identifier and the origin an access key. The second one are the criteria to find the matching origin and target. For example, the target and the origin of the referential constraint must have the same type and the same length.

When all those rules are defined, the referential constraints assistant propose a list of possible foreign key (list of couples) and the analyst can chose those he wants to create.

This assistant is divided into two dialog boxes. In the first one the user can define rules to find target and origin and matching couples are displayed. This dialog box is also used to create the referential constraints and to open the second dialog box in which the matching rules are defined.

A.7.2. Choosing a strategy



1. The entity type parent of the current target group.
2. The attribute of the target entity type.
3. The target groups (prefixed by their entity type).
4. If checked, the attributes are displayed.
5. The entity type parent of the current origin group.
6. The attribute of the origin entity type.
7. The list of the matching groups (prefixed by their entity type).
8. The arrow that represents the foreign key.
9. Opens the dialog box to chose the matching rules.
10. Remove the selected foreign key from the list of the possible foreign keys.
11. Creates the selected foreign key.
12. Resets the matching rules to their default rules.
13. Show all the group couples that match the matching rules.
14. Creates all the proposed foreign keys.
15. Execute a *Voyager2* procedure to create the proposed foreign keys.
16. If checked, marks the origin of the created referential constraint.
17. The type of foreign key to be created.
18. Shows the target entity type into the schema.
19. Shows the origin entity type into the schema.
20. The type of the candidate target groups.
21. The type of the candidate origin groups.

FIGURE 181. Referential constraint ends selection rules.

To use the reference constraint assistant, use the command **Assist/Referential key**.

The radio buttons, on the left, are used to define the target and the origin of the referential constraint. The different possibilities for the target are:

- **Prim. id** The target group must be a primary identifiers of the schema.
- **Any id** The target group must be an identifiers of the schema.
- **Any group** The target group can be any group of the schema.
- **Selected** The target group is a selected groups of the schema.
- **Marker** The target group is a marked groups of the schema.

And for the origin:

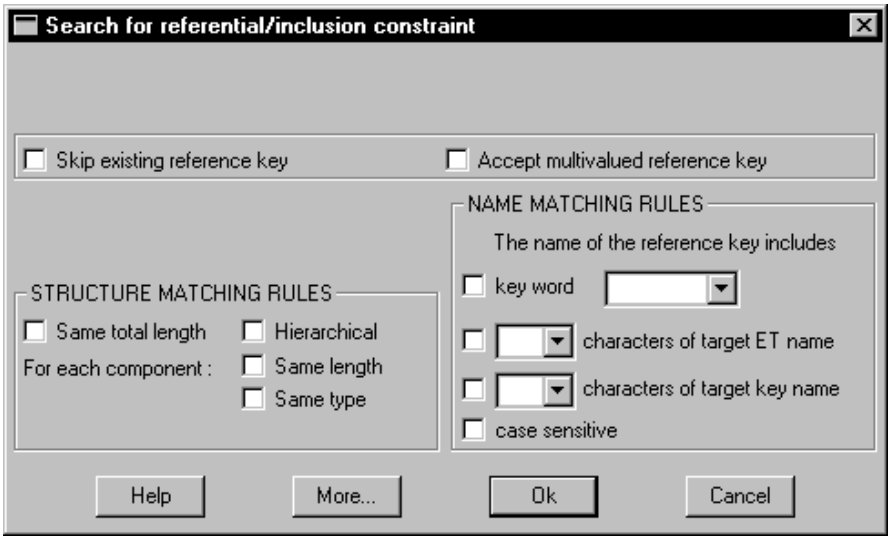
- **Att or group** The origin of the referential constraint can be any attribute (or set of attributes) or any group.
- **Any group** The origin group can be any group of the schema.
- **Selected** The origin group is a selected groups of the schema.
- **Marker** The origin group is a marked groups of the schema.

If the target is **Selected** or **Marked**, then the origin could not be **Selected** or **Marked**. If the origin is **Selected** or **Marked**, then the origin could be **Selected** or **Marked**.

The button **Matching** is used to set the group matching rules. The **Reset** comeback to the default matching rules (each component of the group must have the same type and the same length, does not accept attribute, does not accept multivalued reference key, no name matching rules).

The middle parts displays the proposed referential constraints. The top part contains the target and the bottom the origin of the referential constraint.

A.7.3. The matching rules



1. If checked, the existing referential constraint are not displayed, otherwise they are displayed followed by a "*".
2. If checked, accepts groups that contain a multivalued attribute.
3. Both group must have the same total length (the sum of the length of each components).
4. Groups can contains role. The length of the role is equal to the length of identifier of the entity type connected to the role. This constraint is always associate with the constraint **Same total length**.
5. The components of both groups must have the same length. If the groups have more than one component, they are compared in the order, i.e. the first one with the first one, the second one with the second, and so on.
6. The component of both groups must have the same type. If the groups have more than one component, they are compared in order
7. The name of the reference key must contains the keyword.
8. The name of the reference key must contains (some or all) the characters of the target entity type name.
9. The name of the reference key must contains (some or all) the characters of the target identifier name.
10. The constraints on the name of the attribute of the groups (the group have only one attribute).
11. If checked, the name matching rules are case sensitive.
12. To get some help.
13. To define a Voyager matching rule.

FIGURE 182. Matching rules dialog box.

The search criteria dialog box is obtained by pressing the **Matching** button in the referential constraint assistant dialog box.

The different criteria are:

- *Skip existing reference key*: the group is not selected if the origin group is already the origin of a reference constraint.
- *Structure matching rules*: the two groups must have the same length (**same total length** is checked*)* or each of the groups components must have the same length (**same length** is checked) and/or the same type (**Same type** is checked) or no constraint on the structure (nothing checked). If **hierarchical** is checked (**Same total length** is also checked) and if a group contains a role then the length of the group is the length of the attributes of the group plus the length of the primary identifier of the entity type connected by the role (if the role is multi-domains then it is the maximum of the length of the primary identifier of the entity types connected by the role).
- *Accept multivalued reference key* (only if we are looking for the origin group): accept groups that contain a multivalued attribute.
- *Name matching rules*: this rule contains three criteria on the name of the attributes. If this rules is used the origin group must contains only one attribute.
    - *Key word*: the origin attribute must contains a key word. If this rule is used, the two other name matching rules are applied on the attribute name without the key word.
    - Some or all characters of the origin attribute must be included in the target entity type name.
    - Some or all characters of the origin attribute must be included in the target attribute name.
    - For the two last rules we can choose that all the characters must be included and they must be contiguous. If we choose some (a number, i) then the first i characters of the target entity type or attribute name must be included in the origin attribute, but not necessary in a continuous manner. For example: the three first characters of ABCD are included into FABCDE and in AFBCE but not in CBADE.
    - *case sensitive*: if checked, the name matching rules are case sensitive.
- *More*: the user can gives two Voyager 2 functions, one that checks if two groups are matching and the other that checks if an attribute matches with the target group (see "voyager matching group procedures).

Click on **Ok** to accept the matching rules and to comeback the previous dialog box.

A.7.4. Create the referential constraints

Select the type of the constraint to be created (figure 181) (17). In this version, only **ref** (referential constraint) and **ref equ** (equality constraint) are possible. If **Mark ref** is checked the origin group of the referential constraint will be marked.

Click on the **Create** button to create the current referential constraint. To create all the proposed referential constraint, click on the **Create all** button.

The **Advanced** button can be used to give a *Voyager2* procedure that is called for each proposed referential constraint. This procedure can be used to create the referential constraint into the schema, to print some report or any other function.

A.7.5. Go to the schema

There is a **Goto** button at the right of the target and origin entity type. If you click on one of them, then the corresponding entity type is displayed in the middle of the schema windows.

A.7.6. Changing the selected group

If one of the **Selected** radio buttons is checked, the list of group can be change by selecting other groups in to the schema window without closing the reference constraint assistant. Just select the groups in the schema. Then re-activate the reference constraint assistant and the list of selected groups is updated and the matching group are also displayed, using the criteria as set before

A.7.7. Removing a group/attribute from the list of matching groups

If the matching rules found a (or more) matching group that you do not want to be in the list of matching groups, you can remove it. To remove it, select it and click on the **Remove** button. It disappear and if you click on the **Create all** or **Advanced** button, this referential constraint will not be created.

To redisplay all the groups that have been removed from the matching groups, click on the **Clear** button.

A.7.8. Voyager matching group functions

The user can write its own matching function to check if the two ends of a referential constraint match. The function receives two input parameters, the two ends of the referential constraint. The first one, the target of the referential constraint, is always a group, while the second, the origin, can be a group or an attribute. the user has to write two matching functions. The first one checks if two groups match. The second one checks if the origin attribute match with the target group. The signature of the two functions are the following:

```
export function integer <match_group>(group: <origin_gr>,
                  group: <target_gr>)
```

where

- `<origin_gr>` is the origin group of the referential constraint
- `<target_gr>` is the target group of the referential constraint

and

```
export function integer <match_att_gr>(attribute: <orig_att>,
          group: <target_gr>)
```

where

- `<orig_att>` is the origin attribute of the referential constraint
- `<target_gr>` is the target group of the referential constraint

A.7.9. Example of voyager matching functions

In the name matching rules, there is two rules that check if all or some of the characters of the origin attribute are included into the target attribute or entity type name. This example shows the voyager functions that check if all or some of the characters of the origin attribute are included into the name Two functions `FK_name_in_coll` and `FK_name_in_coll_att` are called through the **more** button of the "matching rules" dialog

```
export function integer FK_name_in_coll(group : org_gr,
 group : targ_gr)
/* Checks that the name of the origin group's (org_gr) attribute
     contains the name of one of the target group's (targ_gr)
     collection*/
  list : l_comp;
  real_component : rc;
  component : co;
  attribute : org_att;
{
// The list of org_gr component
  l_comp := REAL_COMPONENT[rc]
         {REAL_COMP:COMPONENT[co]{@GR_COMP:[org_gr]}};

  if(Length(l_comp) <> 1) then
  {
// If org_gr contains more than one component
    return(0);
  }

  if((GetType(GetFirst(l_comp)) <> SI_ATTRIBUTE)
     and ((GetType(GetFirst(l_comp)) <> CO_ATTRIBUTE))) then
  {
// If the component of 'org_gr' is not an attribute
    return(0);
  }

  org_att := GetFirst(l_comp);
  return(FK_name_in_coll_att(org_att, targ_gr));
}


export function integer FK_name_in_coll_att(attribute : org_att,
     group : targ_gr)
/* Checks that the name of the origin attribute (org_att) contains
     the name of one of the target collections (targ_gr)*/
  list : l_comp;
  real_component : rc;
  component : co;
  attribute : targ_att, att;
  owner_of_att : targ_owner;
  entity_type : targ_et;
  collection : targ_coll;
  coll_et : col_et;
{
// The list of 'targ_gr' component
  l_comp :=
     REAL_COMPONENT[rc]{REAL_COMP:COMPONENT[co]{@GR_COMP:[targ_g
     r]}};

  if(Length(l_comp) <> 1) then
  {
// If 'targ_gr' contains more than one component
    return(0);
  }
  if((GetType(GetFirst(l_comp)) <> SI_ATTRIBUTE)
     and ((GetType(GetFirst(l_comp)) <> CO_ATTRIBUTE))) then
  {
// If the component of 'targ_gr' is not an attribute
    return(0);
```

```
  }

  targ_att := GetFirst(l_comp);

// Search the entity type containing the attribute
  targ_owner :=
       GetFirst(OWNER_OF_ATT[targ_owner]{OWNER_ATT:[targ_att]});
  while(GetType(targ_owner) <> ENTITY_TYPE) do
  {
    att := targ_owner;
    targ_owner :=
            GetFirst(OWNER_OF_ATT[targ_owner]{OWNER_ATT:[att]});
  }
  targ_et := targ_owner;

  for targ_coll in
     COLLECTION[targ_coll]{COLL_COLET:COLL_ET[col_et]{@ENTITY_CO
     LET:[targ_et]}} do
  {
    if(StrFindSubStr(org_att.name, 0, targ_coll.name) >= 0) then
    {
      return(1);
    }
  }
  return(0);
}
```

A.7.10.Voyager "Advanced" procedures

The user can write its own advanced procedure to create the referential constraint or print some report. This procedure receives three input parameters, both ends of the referential constraint and the type of the referential constraint. The first one, the origin of the referential constraint, can be a group or an attribute. The second, the target, is always a group. The user has to write two procedures. The voyager procedures must have the following signature.

```
export procedure <proc_name>(group: <origin_gr>,
     group: <target_gr>, integer: <t>)
```

where

- `<origin_gr>` is the origin group of the referential constraint
- `<target_gr>` is the target group of the referential constraint
- `<t>` is the type of the referential constraint

and

```
export procedure <proc_name>(attribute: <origin_att>,
     group: <target_gr>, integer: <t>)
```

where

- `<origin_att>` is the origin attribute of the referential constraint
- `<target_gr>` is the target group of the referential constraint
- `<t>` is the type of the referential constraint

This procedure is executed for each matching referential constraint. To select the procedure, use the **Browse** button to select the `oxo` file and then select the procedures name in the combo box.

A.7.11.Example of voyager referential keys creation procedures

In fact, the "Advance procedures" rarely create the referential constraints, this is done by the built-in function **create all**. Usually these procedures are used to generate reports or some validation scripts.

In this example, SQL queries are generated to verify that the data verify the proposed referential constraints, i.e. count the number of values of the reference attribute that are not present into the list of the value of the target identifier. For example, if there is a proposed foreign key from `A.A1` to `B.B2`, it generates the following query:

```
select count(*)
from B
where B2 not in (select A1
    from A1);
```

When this query is executed, if the result is equal to 0 then the referential constraint is validated otherwise it is not a referential constraint.

Two validation procedures are needed, `validate_sql` and `validate_sql_att`, that are called through the **Advance** button.

```
export procedure validate_sql(group : gr_org, group :
     gr_targ,
      integer : t)
attribute : att_org, att_targ;
entity_type : ent_org, ent_targ;
data_object : d_o;
real_component : rc;
component : co;
{
  ent_org := GetFirst(DATA_OBJECT[d_o]{DATA_GR : [gr_org]});
  ent_targ := GetFirst(DATA_OBJECT[d_o]{DATA_GR :
     [gr_targ]});
  att_org := GetFirst(REAL_COMPONENT[rc]{REAL_COMP:
                   COMPONENT[co]{@GR_COMP : [gr_org]}});
  att_targ := GetFirst(REAL_COMPONENT[rc]{REAL_COMP:
                   COMPONENT[co]{@GR_COMP : [gr_targ]}});

  SetPrintList("","","");

  print(["select count(*)\n from ",
          ent_org.name, "\nwhere ", att_org.name,
          " not in \n (select ", att_targ.name, "\nfrom ",
          ent_targ.name, ");\n"]);
}


export procedure validate_sql_att(attribute : att_org,
          group : gr_targ, integer : t)
attribute : att_targ, att;
data_object : ent_org, ent_targ;
data_object : d_o;
real_component : rc;
component : co;
owner_of_att : owner;
{
  ent_targ := GetFirst(DATA_OBJECT[d_o]{DATA_GR :
     [gr_targ]});

  owner := GetFirst(OWNER_OF_ATT[owner]{OWNER_ATT :
     [att_org]});
  while((GetType(owner)<> ENTITY_TYPE)
          and (GetType(owner) <> REL_TYPE))
  do {
```

```
    att := owner;
    owner :=GetFirst(OWNER_OF_ATT[owner]{OWNER_ATT :
      [att]});
  }
  ent_org := owner;

  att_targ := GetFirst(REAL_COMPONENT[rc]{REAL_COMP:
                  COMPONENT[co]{@GR_COMP : [gr_targ]}});

  SetPrintList("","","");

  print(["select \"constraint from ", ent_org.name, ".",
         att_org.name, " to ", ent_targ.name, ".",
         att_targ.name, " : \", count(*)\n from ",
         ent_org.name, "\nwhere ", att_org.name,
         " not in \n (select ", att_targ.name, "\nfrom ",
         ent_targ.name, ");\n"]);
}
```

A.8. Miscellaneous Voyager2 programs

This section describes various *Voyager2* program that can be used during various DBRE projects. Their source code and the oxo files (executable through the DB-MAIN **File/Execute Voyager** command) can be found at http://www.info.fundp.ac.be/cgi-bin-dbm/library.

A.8.1. Foreign key analysis

program: eval_fk.oxo

This program displays all the referential constraint of the current schema and says if the origin and the target have the same length.

A.8.2. lexical

program: lexical.oxo

A.8.2.1. Principle

A lexicon is a consistent set of names, each one being assigned to one object of the schema.

The objects affected by this program are: the schema, the entity types, the relationship types, the attributes, the roles and the collections.

A.8.2.2. Usage

The program offers two choice: copy the name of the objects of the current schema into the lexicon or to use the names stored into the lexicon to rename the objects of the current schema.

The first choice displays the existing lexicons and asks if we want to change an existing lexicon or to create a new one. If a new one is created, its name is asked.

The second option displays the existing lexicons. Chose one of them to use it to rename the objects of the schema.

A.8.2.3. Implementation

The schema has a dynamic property "`list_lex`" (string and multivalued) that contains the list of the name of the existing lexicons in this schema. The objects (schema, entity types, relationship types, attributes, roles and collections) have a dynamic property "`lexicon`" (string and multivalued) that contains the object's name for the different lexicons. The ith element of the dynamic property "`lexicon`" of an object is the name according to the lexicon that has the ith element of "`list_lex`" as name.

A.8.3. Compute the physical length

Program: log_phys.oxo

This program compute the physical length (in byte) of each simple attribute, the physical length is stored into the dynamic property '`phys_len`'.

The physical length of an attribute may differ from one DBMS to the other. To allow the use of this tool for an DBMS, some parametrization of the translation rule from the logical length into the physical one are needed [Delvaux-1996]. This transformation is expressed through a formula that associate to each type of data a function that specify how to compute the corresponding physical length.

This function can be express as "for data type X, there is a set of couple composed of a range and a linear function to use if the logical length of the attribute is inside this range".

Figure 183 show an example of such translation rules.

```
boolean : if l in [0..N] then l bits
date :    if l in [0..N] then 12 bits
char :    if l in [0..N] then 8 bits + l * 8 bits
integer : if l in [0..2] then 8 bits
          if l in [3..4] then 16 bits
          if l in [5..9] then 32 bits
          if l in [10..N] then l * 4 bits
float :   if l in [0..N] then 4 bytes
varchar : if l in [0..64] then 1 byte + l * 1 bytes
```

FIGURE 183. An transformation rules example.

Figure 184 table gives the physical length value computed with those formulas.

Type	Logical size	Physical size
boolean	7	7 bits
date	n. a.	12 bits
char	25	208 bits
integer	6	32 bits
integer	9	32 bits
integer	25	100 bits
varchar	64	65 bytes
varchar	65	67 bytes

FIGURE 184. Translation example from logical to physical length.

The syntax of the translation rules are the following:

```
<valid text> ::= "begin physinfo" <type_for>* "end physinfo"
<type_for> ::= "type" <att_type> "{" <rule>* "}" ";"
<att_type> ::= "numeric" | "char" | "boolean" | "date" | "varchar" |
"float"
<rule> ::= <un_rule> | <co_rule> | <lin_rule>
<un_rule> ::= "unit :" <unit> ";"
<si_rule> ::= <lin_rule> | <cst_rule>
<cst_rule> ::= "constant :" <integer> ";"
<lin_rule> ::= "linear :" <integer> ";"
<co_rule> ::= <cst_rule> "level :" "[" <integer> .. <integer> "]"
              "{" <si_rule>* "}" ";"
<unit> ::= "bit" | "byte" | "word"
<integer> ::= <digit>*
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

A *byte* contains 8 *bits* and the *word* contains 32 bits. The value "n" represent the infinity and the following restrictions apply:

It could not have any overlay in the range of a given type.
There is only one unity per type.

Figure 185 gives the formula for the previous example.

```
begin physinfo

type boolean {
  unit : bit;
  linear : 1;
};

type float {
  constant : 4;
};

type date {
  unit : bit;
  constant : 12;
};

type char {
  constant : 1;
  linear : 1;
};

type varchar {
  constant : 1;
  level : [0..64] {
    linear : 1;
  };
  level : [65..n] {
    constant : 1;
    linear : 1;
  };
};

type numeric {
  unit : bit;
  level : [0..2] {
    constant : 8;
  };
  level : [3..4] {
    constant : 16;
  };
  level : [5..9] {
    constant : 32;
  };
  level : [10..n] {
    linear : 4;
  };
};
end physinfo
```

FIGURE 185. The formula for the figure 183 example.

A.8.4. Objects position

Program: pos.oxo

This program has two options :

1. It copies the graphical positions of entity type, relationship type, role, processing unit, collection into the meta-properties 'pos_x', 'pos_y'.
2. It copies the meta-properties 'pos_x' and 'pos_y' as the graphical positions of entity type, relationship type, role, processing unit, collection.

This program can be useful to store the graphical position if the object to allow the analyst to came back to previous positions. Or it can be used to change the position in the meta-properties (manually or through a program) and then apply those position modification.

A.8.5. Report generation

A.8.5.1. RTF

Program: rtf.oxo

This program generates a report in RTF describing the current schema. This report use style sheet, so the user can easily customize the report's presentation by modifying the style sheet.

A.8.6. SQL Validation queries generation

Program: validate_sql.oxo

This program generates SQL queries to validate the current schema (verifies if the constraints are not violated by the data). It validates the foreign keys, the identifier, the 'not null' and computed foreign keys (the 'where' clause must be in the technical description of the origin group).

A.8.7. COBOL validation programs generation

Program: val_fk_cobol.oxo

This program generates a COBOL program the validate each foreign key of the current schema: for each foreign key it count the number of data that violated it and the number of origin records.

A.8.8. Referential key assistant complements

The referential key assistant offers two possibilities to extend it through *Voyager2* procedures and functions. The first one is the one associated with the **More** button of the search dialog box. This allows the user to add its own functions to check if two groups (or a group and an attribute match. The other one allows to declare creation (or generation) procedure through the **Advanced** button. Those procedures receive the origin (group or attribute) and the target (group) of a potential foreign key, so they can create them or generate some reports.

A.8.8.1. Matching functions

A. Foreign key contains the name of the target collection

Program: fk_coll.oxo

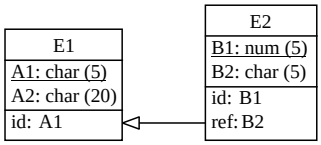
Those functions (`FK_name_in_coll` and `FK_name_in_coll_att`) verify that the name of the origin attribute (the origin group is composed of only one attribute) contains the name of the collection that contains the target entity type.

A.8.8.2. Generation procedures

A. Generate the validation SQL queries

Program: fk_sql.oxo

Those procedures generate for each proposed foreign key a query that count the number of row of the origin table that violate the foreign key.




```
select count(*)
 from E2
 where B2 not in
   (select A1 from E1);
```

FIGURE 186. A foreign key and the validation query.

For example, if the proposed foreign key is the one display in figure 186.a then the procedure will generate the figure 186.b query. If the result of the execution of the query is 0, then all the rows of B respect the foreign key.

B. Generate a report of all the proposed foreign keys

Program: fk_report.oxo

Those procedures generate for each proposed foreign key a report giving the foreign key and the length and type of each of its components.

For example, if the proposed foreign key is the one displayed in figure 186.a, then the procedures will generate the following report

```
#FK#:E2.(E2.B2)-->E1.(E1.A1)
  E2.B2 char(5)
  E1.A1 char(5)
```

C. Creation of the accepted foreign keys

Program: create_fk.oxo

The purpose of the previous report is to allow the analyst to analyze it to validate the proposed foreign keys. When he has validate the foreign keys, the create_fk.oxo program can be used to read the report and to create the valid foreign keys.

In the previous report, each foreign key description is prefixed by the characters "#FK#". If the analyst does not want to create a foreign key, he change the prefix and create_fk.oxo will only create the foreign keys with the original prefix.

ANNEX B *Source code*

B.1. Order.cob

```
 1
 2 IDENTIFICATION DIVISION.
 3 PROGRAM-ID. C-ORD.
 4 ENVIRONMENT DIVISION.
 5 INPUT-OUTPUT SECTION.
 6 FILE-CONTROL.
 7    SELECT CUSTOMER
          ASSIGN TO "CUSTOMER.DAT"
 8        ORGANIZATION IS INDEXED
 9        ACCESS MODE IS DYNAMIC
10        RECORD KEY IS CUS-CODE.
11    SELECT ORDERS
          ASSIGN TO "ORDER.DAT"
12        ORGANIZATION IS INDEXED
13        ACCESS MODE IS DYNAMIC
14        RECORD KEY IS ORD-CODE
15        ALTERNATE RECORD KEY IS
16    ORD-CUSTOMER WITH
          DUPLICATES.
17    SELECT STOCK
          ASSIGN TO "STOCK.DAT"
18        ORGANIZATION IS INDEXED
19        ACCESS MODE IS DYNAMIC
20        RECORD KEY IS STK-CODE.
21
22 DATA DIVISION.
23 FILE SECTION.
24 FD CUSTOMER.
25 01 CUS.
26    02 CUS-CODE PIC X(12).
27    02 CUS-DESCR PIC X(80).
28    02 CUS-HIST PIC X(1000).
29
30 FD ORDERS.
31 01 ORD.
32    02 ORD-CODE PIC 9(10).
33    02 ORD-CUSTOMER PIC X(12).
34    02 ORD-DETAIL PIC X(200).
35
36 FD STOCK.
37 01 STK.
38    02 STK-CODE PIC 9(5).
39    02 STK-NAME PIC X(100).
40    02 STK-LEVEL PIC 9(5).
41
42 WORKING-STORAGE SECTION.
43 01 DESCRIPTION.
44    02 NAME PIC X(20).
45    02 ADDR PIC X(40).
46    02 FUNCT PIC X(10).
47    02 REC-DATE PIC X(10).
48
49 01 LIST-PURCHASE.
50    02 PURCH OCCURS 100 TIMES
          INDEXED BY IND.
51       03 REF-PURCH-STK PIC
         9(5).
52       03 TOT PIC 9(5).
53
54 01 LIST-DETAIL.
55    02 DETAILS OCCURS 20 TIMES
          INDEXED BY IND-DET.
56       03 REF-DET-STK PIC 9(5).
57       03 ORD-QTY PIC 9(5).
58
59 01 CHOICE PIC X.
60 01 END-FILE PIC 9.
61 01 END-DETAIL PIC 9.
62 01 EXIST-PROD PIC 9.
63 01 PROD-CODE PIC 9(5).
64
65 01 TOT-COMP PIC 9(5) COMP.
```

```
01 QTY PIC 9(5) COMP.
01 NEXT-DET PIC 99.

PROCEDURE DIVISION.
MAIN.
  PERFORM INIT.
  PERFORM PROCESS
   UNTIL CHOICE = 0.
  PERFORM CLOSING.
  STOP RUN.

INIT.
  OPEN I-O CUSTOMER.
  OPEN I-O ORDERS.
  OPEN I-O STOCK.

PROCESS.
  DISPLAY "1 NEW CUSTOMER".
  DISPLAY "2 NEW STOCK".
  DISPLAY "3 NEW ORDER".
  DISPLAY "4 LIST OF
   CUSTOMERS".
  DISPLAY "5 LIST OF STOCKS".
  DISPLAY "6 LIST OF ORDERS".
  DISPLAY "0 END".
  ACCEPT CHOICE.
  IF CHOICE = 1
    PERFORM NEW-CUS.
  IF CHOICE = 2
    PERFORM NEW-STK.
  IF CHOICE = 3
    PERFORM NEW-ORD.
  IF CHOICE = 4
    PERFORM LIST-CUS.
  IF CHOICE = 5
    PERFORM LIST-STK.
  IF CHOICE = 6
    PERFORM LIST-ORD.

CLOSING.
  CLOSE CUSTOMER.
  CLOSE ORDERS.
  CLOSE STOCK.

NEW-CUS.
  DISPLAY "NEW CUSTOMER :".
  DISPLAY "CUSTOMER CODE ?"
    WITH NO ADVANCING.
  ACCEPT CUS-CODE.

  DISPLAY "NAME DU CUSTOMER : "
    WITH NO ADVANCING.
  ACCEPT NAME.
  DISPLAY "ADDR OF CUSTOMER:"
    WITH NO ADVANCING.
  ACCEPT ADDR.
  DISPLAY "FUNCT. OF CUSTOMER:"
    WITH NO ADVANCING.
  ACCEPT FUNCT.
  DISPLAY "DATE : "
   WITH NO ADVANCING.
  ACCEPT REC-DATE.
  MOVE DESCRIPTION TO CUS-
   DESCR.
  PERFORM INIT-HIST.
  WRITE CUS
    INVALID KEY DISPLAY
   "ERROR".

LIST-CUS.
  DISPLAY "LISTE DES
   CUSTOMERS".
  CLOSE CUSTOMER.
  OPEN I-O CUSTOMER.
  MOVE 1 TO END-FILE.
  PERFORM READ-CUS
   UNTIL (END-FILE = 0).

READ-CUS.
  READ CUSTOMER NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      DISPLAY CUS-CODE
      DISPLAY CUS-DESCR
      DISPLAY CUS-HIST.

NEW-STK.
  DISPLAY "NEW STOCK".
  DISPLAY "PRODUCT NUMBER : "
    WITH NO ADVANCING.
  ACCEPT STK-CODE.

  DISPLAY "NAME : "
   WITH NO ADVANCING.
  ACCEPT STK-NAME.

  DISPLAY "LEVEL : "
   WITH NO ADVANCING.
  ACCEPT STK-LEVEL.

  WRITE STK
    INVALID KEY DISPLAY"ERROR".

LIST-STK.
  DISPLAY "LIST OF STOCKS ".

  CLOSE STOCK.
  OPEN I-O STOCK.
```

```
165
166   MOVE 1 TO END-FILE.
167   PERFORM READ-STK
          UNTIL END-FILE = 0.
168
169 READ-STK.
170   READ STOCK NEXT
171     AT END MOVE 0 TO END-FILE
172     NOT AT END
173       DISPLAY STK-CODE
174       DISPLAY STK-NAME
175       DISPLAY STK-LEVEL.
176
177 NEW-ORD.
178   DISPLAY "NEW ORDER".
179   DISPLAY "ORDER NUMBER : "
180     WITH NO ADVANCING.
181   ACCEPT ORD-CODE.
182
183   MOVE 1 TO END-FILE.
184   PERFORM READ-CUS-CODE
       UNTIL END-FILE = 0.
185   MOVE CUS-DESCR TO
       DESCRIPTION.
186   DISPLAY NAME.
187   MOVE CUS-CODE
        TO ORD-CUSTOMER.
188   MOVE CUS-HIST
        TO LIST-PURCHASE.
189
190   SET IND-DET TO 1.
191   MOVE 1 TO END-FILE.
192   PERFORM READ-DETAIL
193     UNTIL END-FILE = 0
                OR IND-DET = 21.
194   MOVE LIST-DETAIL
        TO ORD-DETAIL.
195
196   WRITE ORD
197     INVALID KEY
          DISPLAY "ERROR".
198
199   MOVE LIST-PURCHASE
200     TO CUS-HIST.
201   REWRITE CUS
202     INVALID KEY
          DISPLAY "ERROR CUS".
203 READ-CUS-CODE.
204   DISPLAY "CUSTOMER NUMBER : "
205     WITH NO ADVANCING.
206   ACCEPT CUS-CODE.
207   MOVE 0 TO END-FILE.
208   READ CUSTOMER INVALID KEY
209     DISPLAY "NO SUCH CUSTOMER"
210     MOVE 1 TO END-FILE
211   END-READ.
212
213 READ-DETAIL.
214   DISPLAY "PRODUCT CODE
       (0=END)".
215   ACCEPT PROD-CODE.
216   IF PROD-CODE = 0
217     MOVE 0
218       TO REF-DET-STK(IND-DET)
219     MOVE 0 TO END-FILE
220   ELSE
221     PERFORM READ-PROD-CODE.
222
223 READ-PROD-CODE.
224   MOVE 1 TO EXIST-PROD.
225   MOVE PROD-CODE TO STK-CODE.
226   READ STOCK INVALID KEY
227     MOVE 0 TO EXIST-PROD.
228   IF EXIST-PROD = 0
229     DISPLAY "NO SUCH PRODUCT"
230   ELSE
231     PERFORM UPDATE-ORD-DETAIL.
232
233 UPDATE-ORD-DETAIL.
234   MOVE 1 TO NEXT-DET.
235   DISPLAY "QUANTITY ORDERED : "
236     WITH NO ADVANCING
237   ACCEPT ORD-QTY(IND-DET).
238   PERFORM UNTIL
239     (NEXT-DET < IND-DET
240      AND REF-DET-STK(NEXT-DET)
             = PROD-CODE)
241     OR IND-DET = NEXT-DET
242     ADD 1 TO NEXT-DET
243   END-PERFORM.
244   IF IND-DET = NEXT-DET
245     MOVE PROD-CODE
246     TO REF-DET-STK(IND-DET)
247     PERFORM UPDATE-CUS-HIST
248     SET IND-DET UP BY 1
249   ELSE
250     DISPLAY "ERROR: ALREADY
                ORDERED".
251
252 UPDATE-CUS-HIST.
253   SET IND TO 1.
254   PERFORM UNTIL
255     REF-PURCH-STK(IND) = PROD-
       CODE
256     OR REF-PURCH-STK(IND) = 0
257     OR IND = 101
258       SET IND UP BY 1
```

```
259   END-PERFORM.
260   IF IND = 101
261     DISPLAY "ERR : HISTORY
       OVERFLOW"
262     EXIT.
263   IF REF-PURCH-STK(IND)
264        = PROD-CODE
265     ADD ORD-QTY(IND-DET) TO
       TOT(IND)
266   ELSE
267     MOVE PROD-CODE
268          TO REF-PURCH-STK(IND)
269     MOVE ORD-QTY(IND-DET)
                TO TOT(IND).
270
271 LIST-ORD.
272   DISPLAY "LIST OF ORDERS ".
273   CLOSE ORDERS.
274   OPEN I-O ORDERS.
275   MOVE 1 TO END-FILE.
276   PERFORM READ-ORD
       UNTIL END-FILE = 0.
277
278 READ-ORD.
279   READ ORDERS NEXT
280     AT END MOVE 0 TO END-FILE
281     NOT AT END
282       DISPLAY "ORD-CODE "
283         WITH NO ADVANCING
284       DISPLAY ORD-CODE
285       DISPLAY "ORD-CUSTOMER "
286         WITH NO ADVANCING
287       DISPLAY ORD-CUSTOMER
288       DISPLAY "ORD-DETAIL "
289       MOVE ORD-DETAIL
                TO LIST-DETAIL
290       SET IND-DET TO 1
291       MOVE 1 TO END-DETAIL
292       PERFORM DISPLAY-DETAIL.
293
294 INIT-HIST.
295   SET IND TO 1.
296   PERFORM UNTIL IND = 100
297     MOVE 0
          TO REF-PURCH-STK(IND)
298     MOVE 0 TO TOT(IND)
299     SET IND UP BY 1
300   END-PERFORM.
301   MOVE LIST-PURCHASE
          TO CUS-HIST.
302
303 DISPLAY-DETAIL.
304   IF IND-DET = 21
305     MOVE 0 TO END-DETAIL
306     EXIT.
307   IF REF-DET-STK(IND-DET) = 0
308     MOVE 0 TO END-DETAIL
309   ELSE
310     DISPLAY
             REF-DET-STK(IND-DET)
311     DISPLAY ORD-QTY(IND-DET)
312     SET IND-DET UP BY 1.
```

B.2. Validation program (automatically generated)

```
1   *
2   *  Date : 9-1-2003, 17 : 18 : 45
3   *  Author : Jean Henrard
4   *  Program generated by val_fk_cobol.oxo
5       IDENTIFICATION DIVISION.
6       PROGRAM-ID. physical.
7
8       ENVIRONMENT DIVISION.
9       INPUT-OUTPUT SECTION.
10      FILE-CONTROL.
11          SELECT STOCK ASSIGN TO "STOCK"
12              ORGANIZATION IS INDEXED
13              ACCESS MODE IS DYNAMIC
14              RECORD KEY IS STK-CODE.
15
16          SELECT ORDERS ASSIGN TO "ORDERS"
17              ORGANIZATION IS INDEXED
18              ACCESS MODE IS DYNAMIC
19              RECORD KEY IS ORD-CODE
20              ALTERNATE RECORD KEY IS ORD-CUSTOMER
21                  WITH DUPLICATES.
22
23          SELECT CUSTOMER ASSIGN TO "CUSTOMER"
24              ORGANIZATION IS INDEXED
25              ACCESS MODE IS DYNAMIC
26              RECORD KEY IS CUS-CODE.
27
28
29      DATA DIVISION.
30      FILE SECTION.
31      FD STOCK.
32      01 STK.
33          02 STK-CODE   PIC 9(5).
34          02 STK-NAME   PIC X(100).
35          02 STK-LEVEL  PIC 9(5).
36      FD ORDERS.
37      01 ORD.
38          02 ORD-CODE   PIC 9(10).
39          02 ORD-CUSTOMER  PIC X(12).
40          02 ORD-DETAIL.
41              03 DETAILS OCCURS 20 TIMES
42                  INDEXED BY IND-DETAILS.
43                      04 REF-DET-STK  PIC 9(5).
44                      04 ORD-QTY   PIC 9(5).
45      FD CUSTOMER.
46      01 CUS.
47          02 CUS-CODE   PIC X(12).
48          02 CUS-DESCR.
49              03 NAME   PIC X(20).
50              03 ADDR   PIC X(40).
51              03 FUNCT  PIC X(10).
52              03 REC-DATE   PIC X(10).
53          02 CUS-HIST.
54              03 PURCH OCCURS 100 TIMES
55                  INDEXED BY IND-PURCH.
56                      04 REF-PURCH-STK  PIC 9(5).
57                      04 TOT  PIC 9(5).
58
59      WORKING-STORAGE SECTION.
60          01 END-FILE PIC 9.
61          01 COUNT-ERROR PIC 9(5).
62          01 COUNT-TOTAL PIC 9(5).
63          01 FK-DESC PIC X(100).
64
65      PROCEDURE DIVISION.
66      MAIN.
67          PERFORM CHECK-CUS-ref.
68          MOVE "CUS:{REF-PURCH-STK}-->STK:{STK-
            CODE}"
69              TO FK-DESC.
70          MOVE 0 TO COUNT-TOTAL.
71          MOVE 0 TO COUNT-ERROR.
```

```
72        PERFORM DISPLAY-REPORT.
73
74        PERFORM CHECK-ORD-ORD-CUSTOMER.
75        MOVE "ORD:{ORD-CUSTOMER}-->CUS:{CUS-CODE}"
76         TO FK-DESC.
77        MOVE 0 TO COUNT-TOTAL.
78        MOVE 0 TO COUNT-ERROR.
79        PERFORM DISPLAY-REPORT.
80
81        PERFORM CHECK-ORD-ref.
82        MOVE "ORD:{REF-DET-STK}-->STK:{STK-CODE}"
83         TO FK-DESC.
84        MOVE 0 TO COUNT-TOTAL.
85        MOVE 0 TO COUNT-ERROR.
86        PERFORM DISPLAY-REPORT.
87
88        STOP RUN.
89
90     DISPLAY-REPORT.
91        DISPLAY FK-DESC.
92        DISPLAY "TOTAL NUMBER OF RECORDS: " COUNT-
    TOTAL.
93        DISPLAY "NUMBER OF ERRORS: " COUNT-ERROR.
94
95     CHECK-CUS-ref.
96       OPEN INPUT CUSTOMER.
97       MOVE 1 TO END-FILE.
98       PERFORM READ-CUS-ref UNTIL END-FILE = 0.
99
100    READ-CUS-ref.
101       READ CUSTOMER NEXT
102         AT END MOVE 0 TO END-FILE
103         NOT AT END
104          PERFORM READ-TARG-CUS-ref.
105
106    READ-TARG-CUS-ref.
107       SET IND-PURCH TO 1.
108       PERFORM READ-TARG-CUS-REF-ARRAY UNTIL IND-
    PURCH = 101.
109
110   READ-TARG-CUS-REF-ARRAY.
111       ADD 1 TO COUNT-TOTAL.
112       MOVE REF-PURCH-STK(IND-PURCH) TO STK-CODE.
113       READ STOCK
114        INVALID KEY MOVE 1 TO COUNT-ERROR.
115       SET IND-PURCH UP BY 1.
116
117    CHECK-ORD-ORD-CUSTOMER.
118      OPEN INPUT ORDERS.
119      MOVE 1 TO END-FILE.
120     PERFORM READ-ORD-ORD-CUSTOMER UNTIL END-FILE
    = 0.
121
122    READ-ORD-ORD-CUSTOMER.
123       READ ORDERS NEXT
124         AT END MOVE 0 TO END-FILE
125         NOT AT END
126          PERFORM READ-TARG-ORD-ORD-CUSTOMER.
127
128    READ-TARG-ORD-ORD-CUSTOMER.
129       ADD 1 TO COUNT-TOTAL.
130       MOVE ORD-CUSTOMER TO CUS-CODE.
131       READ CUSTOMER
132         INVALID KEY MOVE 1 TO COUNT-ERROR.
133
134    CHECK-ORD-ref.
135       OPEN INPUT ORDERS.
136       MOVE 1 TO END-FILE.
137       PERFORM READ-ORD-ref UNTIL END-FILE = 0.
138
139    READ-ORD-ref.
140       READ ORDERS NEXT
```

```
141         AT END MOVE 0 TO END-FILE
142         NOT AT END
143           PERFORM READ-TARG-ORD-ref.
144
145    READ-TARG-ORD-ref.
146       SET IND-DETAILS TO 1.
147       PERFORM READ-TARG-ORD-REF-ARRAY UNTIL IND-
    DETAILS = 101.
148
149    READ-TARG-ORD-REF-ARRAY.
150       ADD 1 TO COUNT-TOTAL.
151       MOVE REF-DET-STK(IND-DETAILS) TO STK-CODE.
152       READ STOCK
153        INVALID KEY MOVE 1 TO COUNT-ERROR.
154       SET IND-DETAILS UP BY 1.
155
156
```

B.3. SQL-DDL code

```
create table CUSTOMER (
    CODE char(12) not null,
    NAME char(20) not null,
    ADDR char(40) not null,
    FUNCT char(10),
    REC_DATE char(10) not null,
    primary key (CODE));

create table DETAIL (
    ORDERS numeric(10) not null,
    PRODUCT numeric(5) not null,
    ORD_QTY numeric(5) not null,
    primary key (PROD, ORDERS));

create table ORDERS (
    CODE numeric(10) not null,
    ORD_DATE char(8) not null,
    CUSTOMER char(12) not null,
    primary key (CODE));

create table PURCH (
    CUSTOMER char(12) not null,
    PRODUCT numeric(5) not null,
    TOT numeric(5) not null,
    primary key (PROD, CUSTOMER));

create table PRODUCT (
    CODE numeric(5) not null,
    NAME char(100) not null,
    PRICE numeric(5) not null,
    primary key (CODE));

create index GRDETAIL
    on DETAIL (PRODUCT);

create index GRORDERS
    on ORDERS (CUSTOMER);

create index GRPURCH
    on PURCH (PRODUCT);
```

```
1     IDENTIFICATION DIVISION.
2     PROGRAM-ID. C-ORD.
3     DATA DIVISION.
4
5     WORKING-STORAGE SECTION.
6    * Copy in the SQL Comm. Area (SQLCA)
7         EXEC SQL INCLUDE SQLCA END-EXEC.
8    * Copy in the Oracle Comm. Area (ORACA)
9         EXEC SQL INCLUDE ORACA END-EXEC.
10
11        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
12    01 CUS.
13      02 CUS-CODE PIC X(12).
14      02 CUS-NAME PIC X(20).
15      02 CUS-ADDR PIC X(40).
16      02 CUS-FUNCT PIC X(10).
17      02 CUS-REC-DATE PIC X(10).
18
19    01 ORD.
20      02 ORD-CODE PIC 9(10).
21      02 ORD-CUSTOMER PIC X(12).
22
23    01 PROD.
24      02 PROD-CODE PIC 9(5).
25      02 PROD-NAME PIC X(100).
26      02 PRICE PIC 9(5).
27
28    01 PURCH.
29      02 PURCH-CUST PIC X(12).
30      02 PURCH-PROD PIC 9(5).
31      02 PURCH-TOT PIC 9(5).
32
33    01 DETAIL.
34      02 DET-CUST PIC X(12).
35      02 DET-PROD PIC 9(5).
36      02 DET-QTY PIC 9(5).
37
38    01  USERNAME       PIC X(20).
39    01  PASSWD         PIC X(20).
40       EXEC SQL END DECLARE SECTION END-EXEC.
41
42    01 CHOICE PIC X.
43    01 END-FILE PIC 9.
44    01 PROD-CODE PIC 9(5).
45
46    PROCEDURE DIVISION.
47    MAIN.
48       PERFORM INIT.
49       PERFORM PROCESS UNTIL CHOICE = 0.
50       STOP RUN.
51
52    INIT.
53       DISPLAY "USERNAME: " WITH NO ADVANCING.
54       ACCEPT USERNAME.
55       DISPLAY "PASSWORD: " WITH NO ADVANCING.
56       ACCEPT PASSWD.
57       EXEC SQL
58        CONNECT :USERNAME IDENTIFIED BY :PASSWD
59       END-EXEC.
60
61    PROCESS.
62       DISPLAY "1 NEW CUSTOMER".
63       DISPLAY "2 NEW PRODUCT".
64       DISPLAY "3 NEW ORDER".
65       DISPLAY "4 LIST OF CUSTOMERS".
66       DISPLAY "5 LIST OF PRODUCTS".
67       DISPLAY "6 LIST OF ORDERS".
68       DISPLAY "0 END".
69       ACCEPT CHOICE.
70       IF CHOICE = 1
71        PERFORM NEW-CUS.
72       IF CHOICE = 2
```

```
73          PERFORM NEW-PROD.
74        IF CHOICE = 3
75          PERFORM NEW-ORD.
76        IF CHOICE = 4
77          PERFORM LIST-CUS.
78        IF CHOICE = 5
79          PERFORM LIST-PROD.
80        IF CHOICE = 6
81          PERFORM LIST-ORD.
82
83     NEW-CUS.
84        DISPLAY "NEW CUSTOMER :".
85        DISPLAY "CUSTOMER CODE ?"
86         WITH NO ADVANCING.
87        ACCEPT CUS-CODE.
88
89        DISPLAY "NAME DU CUSTOMER : "
90         WITH NO ADVANCING.
91        ACCEPT CUS-NAME.
92        DISPLAY "ADDRESS OF CUSTOMER : "
93          WITH NO ADVANCING.
94        ACCEPT CUS-ADDR.
95        DISPLAY "FUNCTION OF CUSTOMER : "
96          WITH NO ADVANCING.
97        ACCEPT CUS-FUNCT.
98        DISPLAY "DATE : " WITH NO ADVANCING.
99        ACCEPT CUS-REC-DATE.
100
101       EXEC SQL
102         INSERT INTO CUSTOMER
103           VALUES (:CUS-CODE, :CUS-NAME, :CUS-ADDR,
104                   :CUS-FUNCT, :CUS-REC-DATE)
105       END-EXEC.
106
107       IF(SQLCODE = 0)
108         EXEC SQL
109           COMMIT
110         END-EXEC
111       ELSE
112         DISPLAY "ERROR".
113
114    LIST-CUS.
115       DISPLAY "LISTS OF THE CUSTOMERS".
116       EXEC SQL
117         DECLARE ALL_CUST CURSOR FOR
118         SELECT CODE, NAME, ADDR, FUNCT, REC_DATE
119         FROM CUSTOMER
120         ORDER BY CODE
121       END-EXEC.
122       EXEC SQL
123        OPEN ALL_CUST
124       END-EXEC.
125       IF(SQLCODE = 0)
126         MOVE 0 TO END-FILE.
127       PERFORM READ-CUS UNTIL (END-FILE = 1).
128
129    READ-CUS.
130       EXEC SQL
131        FETCH ALL_CUST
132        INTO :CUS-CODE, :CUS-NAME, :CUS-ADDR,
133             :CUS-FUNCT, :CUS-REC-DATE
134       END-EXEC.
135       IF(SQLCODE = 0)
136         MOVE 1 TO END-FILE
137         DISPLAY CUS-CODE CUS-NAME
138         DISPLAY CUS-ADDR
139         DISPLAY CUS-FUNCT CUS-REC-DATE
140         PERFORM DISP-CUS-HISTORY.
141
142    DISP-CUS-HISTORY.
143     EXEC SQL
144        DECLARE CUS_HIST CURSOR FOR
145        SELECT P.TOT, PR.NAME
146        FROM PURCH P, PRODUCT PR
```

```
147       WHERE P.PRODUCT = PR.CODE
148         AND P.CUSTOMER = :CUS-CODE
149      END-EXEC.
150      EXEC SQL
151       OPEN CUS_HIST
152      END-EXEC.
153      DISPLAY "PRODUCT    TOTAL".
154
155     PERFORM DISP-HISTORY UNTIL (SQLCODE NOT = 0).
156
157   DISP-HISTORY.
158      EXEC SQL
159       FETCH CUS_HIST
160       INTO :PURCH-TOT, :PROD-NAME
161      END-EXEC.
162      IF(SQLCODE = 0)
163        DISPLAY PROD-NAME PURCH-TOT.
164
165   NEW-PROD.
166      DISPLAY "NEW PRODUCT".
167      DISPLAY "PRODUCT NUMBER : "
168        WITH NO ADVANCING.
169      ACCEPT PROD-CODE.
170
171      DISPLAY "NAME : " WITH NO ADVANCING.
172      ACCEPT PROD-NAME.
173
174      DISPLAY "LEVEL : " WITH NO ADVANCING.
175      ACCEPT PRICE.
176
177      EXEC SQL
178       INSERT INTO PRODUCT
179         VALUES (:PROD-CODE, :PROD-NAME,
180                 :PROD-PRICE)
181      END-EXEC.
182
183      IF(SQLCODE = 0)
184        EXEC SQL
185          COMMIT
186        END-EXEC
187      ELSE
188        DISPLAY "ERROR".
189
190   LIST-PROD.
191      DISPLAY "LIST OF PRODUCTS ".
192      EXEC SQL
193       DECLARE ALL_PROD CURSOR FOR
194       SELECT CODE, NAME, PRICE
195       FROM PRODUCT
196       ORDER BY CODE
197      END-EXEC.
198      EXEC SQL
199       OPEN ALL_PROD
200      END-EXEC.
201
202      PERFORM READ-PROD UNTIL SQLCODE NOT = 0.
203
204   READ-PROD.
205      EXEC SQL
206       FETCH ALL_PROD
207       INTO :PROD-CODE, :PROD-NAME, :PROD-PRICE
208      END-EXEC.
209      IF(SQLCODE = 0)
210        DISPLAY PROD-CODE
211        DISPLAY PROD-NAME
212        DISPLAY PROD-PRICE.
213
214   NEW-ORD.
215      DISPLAY "NEW ORDER".
216      DISPLAY "ORDER NUMBER : "
217        WITH NO ADVANCING.
218      ACCEPT ORD-CODE.
```

```
219
220      MOVE 1 TO SQLCODE.
221      PERFORM READ-CUS-CODE UNTIL SQLCODE = 0.
222      DISPLAY CUS-NAME.
223      EXEC SQL
224       INSERT INTO ORDERS VALUES (:ORD-CODE,
225                                  SYSDATE, :CUS-CODE)
226      END-EXEC.
227      IF(SQLCODE NOT = 0)
228        DISPLAY "ERROR"
229      ELSE
230        MOVE 0 TO END-FILE
231        PERFORM READ-DETAIL
232          UNTIL END-FILE = 1
233
234       EXEC SQL
235         COMMIT
236       END-EXEC.
237
238   READ-CUS-CODE.
239      DISPLAY "CUSTOMER NUMBER : "
240        WITH NO ADVANCING.
241      ACCEPT CUS-CODE.
242      EXEC SQL
243       SELECT CODE
244       INTO :CUS-CODE
245       FROM CUSTOMER
246       WHERE CODE = :CUS-CODE
247      END-EXEC.
248      IF(SQLCODE NOT = 0)
249        DISPLAY "NO SUCH CUSTOMER".
250
251   READ-DETAIL.
252      DISPLAY "PRODUCT CODE (0 = END) : ".
253      ACCEPT PROD-CODE.
254      IF PROD-CODE = 0
255        MOVE 1 TO END-FILE
256      ELSE
257        PERFORM READ-PROD-CODE.
258
259   READ-PROD-CODE.
260      EXEC SQL
261       SELECT CODE INTO :PROD-CODE
262       FROM PRODUCT
263       WHERE CODE = :PROD-CODE
264      END-EXEC.
265      IF SQLCODE = 0
266        PERFORM UPDATE-ORD-DETAIL
267      ELSE
268        DISPLAY "NO SUCH PRODUCT".
269
270   UPDATE-ORD-DETAIL.
271      DISPLAY "QUANTITY ORDERED : "
272        WITH NO ADVANCING
273      ACCEPT DET-QTY.
274      EXEC SQL
275       SELECT ORD-QTY
276       FROM DETAIL
277       WHERE ORDER = :ORD-CODE
278          AND PRODUCT = :PROD-CODE
279      END-EXEC
280
281      IF SQLCODE = 0
282        DISPLAY "ERROR : ALREADY ORDERED"
283      ELSE
284        EXEC SQL
285         INSERT INTO DETAIL
286           VALUES (:ORD-CODE, :PROD-CODE, :DET-QTY)
287        END-EXEC
288        PERFORM UPDATE-CUS-HIST.
289
290   UPDATE-CUS-HIST.
```

```
291      EXEC SQL
292       SELECT TOT INTO :PURCH-TOT
293       FROM PURCH
294       WHERE CUSTOMER = :CUS-CODE
295           AND PRODUCT = :PROD-CODE
296      END-EXEC.
297      IF(SQLCODE = 0)
298        EXEC SQL
299         INSERT INTO PURCH VALUES(:CUS-CODE,
300                                  :PROD-CODE, :DET-QTY)
301        END-EXEC
302      ELSE
303        EXEC SQL
304         UPDATE PRUCH SET
305           TOT = (:PURCH-TOT + :DET-QTY)
306         WHERE CUSTOMER = :CUS-CODE
307           AND PRODUCT = :PROD-CODE
308        END-EXEC.
309
310   LIST-ORD.
311      DISPLAY "LIST OF ORDERS ".
312      EXEC SQL
313       DECLARE ALL_ORD CURSOR FOR
314       SELECT CODE, CUSTOMER
315       FROM ORDERS
316       ORDER BY CODE
317      END-EXEC.
318      EXEC SQL
319       OPEN ALL_ORD
320      END-EXEC.
321      MOVE 0 TO END-FILE
322      PERFORM READ-ORD UNTIL END-FILE = 1.
323
324   READ-ORD.
325      EXEC SQL
326       FETCH ALL_ORD
327        INTO :ORD-CODE, :ORD-CUSTOMER
328      END-EXEC.
329      IF(SQLCODE = 0)
330        MOVE 1 TO END-FILE
331        DISPLAY "ORD-CODE "
332          WITH NO ADVANCING
333        DISPLAY ORD-CODE
334        DISPLAY "ORD-CUSTOMER "
335          WITH NO ADVANCING
336        DISPLAY ORD-CUSTOMER
337        DISPLAY "ORD-DETAIL "
338        EXEC SQL
339         DECLARE ORD_DEF CURSOR FOR
340         SELECT D.ORD-QTY, P.NAME
341         FROM DETAIL D, PRODUCT P
342         WHERE D.ORDER = :ORD-CODE
343           AND D.PRODUCT = P.CODE
344        END-EXEC
345        EXEC SQL
346         OPEN ALL_ORD
347        END-EXEC
348        DISPLAY "PROD NAME    QUANTITY"
349        PERFORM DISPLAY-DETAIL
350           UNTIL SQLCODE NOT = 0.
351
352   DISPLAY-DETAIL.
353      EXEC SQL
354       FETCH ORD_DEF
355         INTO :DET-QTY, :PROD-NAME
356      END-EXEC.
357      DISPLAY PROD-NAME DET-QTY.
358
359
```

2.5. Modified embedded code

```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. C-ORD.
3   DATA DIVISION.
4
5    WORKING-STORAGE SECTION.
6   * Copy in the SQL Comm. Area (SQLCA)
7   *E    EXEC SQL INCLUDE SQLCA END-EXEC.
8   * Copy in the Oracle Comm. Area (ORACA)
9   *E    EXEC SQL INCLUDE ORACA END-EXEC.
10  01 SQLCODE PIC 9(9).
11
12  *E    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
13  01 TAB-CUSTOMER.
14       CUSTOMER--CODE PIC X(12).
15       CUSTOMER--NAME PIC X(20).
16       CUSTOMER--ADDR PIC X(40).
17       CUSTOMER--FUNCT PIC X(10).
18       CUSTOMER--REC-DATE PIC X(10).
19
20  01 TAB-DETAIL.
21       DETAIL--ORDERS PIC 9(10).
22       DETAIL--PRODUCT PIC 9(5).
23       DETAIL--ORD-QTY PIC 9(5).
24
25  01 TAB-ORDERS.
26       ORDERS--CODE PIC 9(10).
27       ORDERS--ORD-DATE PIC X(8).
28       ORDERS--CUSTOMER PIC X(12).
29
30  01 TAB-PURCH.
31       PURCH--CUSTOMER PIC X(12).
32       PURCH--PRODUCT PIC 9(5).
33       PURCH--TOT PIC 9(5).
34
35  01 TAB-PRODUCT.
36       PRODUCT--CODE PIC 9(5).
37       PRODUCT--NAME PIC X(100).
38       PRODUCT--PRICE PIC 9(5).
39
40  01 W-TMP PIC 9(5).
41
42  01 CUS.
43       02 CUS-CODE PIC X(12).
44       02 CUS-NAME PIC X(20).
45       02 CUS-ADDR PIC X(40).
46       02 CUS-FUNCT PIC X(10).
47       02 CUS-REC-DATE PIC X(10).
48
49  01 ORD.
50       02 ORD-CODE PIC 9(10).
51       02 ORD-CUSTOMER PIC X(12).
52
53  01 PROD.
54       02 PROD-CODE PIC 9(5).
55       02 PROD-NAME PIC X(100).
56       02 PROD-PRICE PIC 9(5).
57
58  01 PURCH.
59       02 PURCH-CUST PIC X(12).
60       02 PURCH-PROD PIC 9(5).
61       02 PURCH-TOT PIC 9(5).
62
63  01 DETAIL.
64       02 DET-CUST PIC X(12).
65       02 DET-PROD PIC 9(5).
66       02 DET-QTY PIC 9(5).
67
68  01  USERNAME        PIC X(20).
69  01  PASSWD          PIC X(20).
70  *E    EXEC SQL END DECLARE SECTION END-EXEC.
71
72  01 CHOICE PIC X.
```

```
73   01 END-FILE PIC 9.
74   01 PROD-CODE PIC 9(5).
75
76   PROCEDURE DIVISION.
77   MAIN.
78      PERFORM INIT.
79      PERFORM PROCESS UNTIL CHOICE = 0.
80      STOP RUN.
81
82   INIT.
83      DISPLAY "USERNAME: " WITH NO ADVANCING.
84      ACCEPT USERNAME.
85      DISPLAY "PASSWORD: " WITH NO ADVANCING.
86      ACCEPT PASSWD.
87  *E    EXEC SQL
88  *E     CONNECT :USERNAME IDENTIFIED BY :PASSWD
89  *E    END-EXEC.
90
91   PROCESS.
92      DISPLAY "1 NEW CUSTOMER".
93      DISPLAY "2 NEW PRODUCT".
94      DISPLAY "3 NEW ORDER".
95      DISPLAY "4 LIST OF CUSTOMERS".
96      DISPLAY "5 LIST OF PRODUCTS".
97      DISPLAY "6 LIST OF ORDERS".
98      DISPLAY "0 END".
99      ACCEPT CHOICE.
100     IF CHOICE = 1
101      PERFORM NEW-CUS.
102     IF CHOICE = 2
103      PERFORM NEW-PROD.
104     IF CHOICE = 3
105      PERFORM NEW-ORD.
106     IF CHOICE = 4
107      PERFORM LIST-CUS.
108     IF CHOICE = 5
109      PERFORM LIST-PROD.
110     IF CHOICE = 6
111      PERFORM LIST-ORD.
112
113  NEW-CUS.
114     DISPLAY "NEW CUSTOMER :".
115     DISPLAY "CUSTOMER CODE ?"
116      WITH NO ADVANCING.
117     ACCEPT CUS-CODE.
118
119     DISPLAY "NAME DU CUSTOMER : "
120      WITH NO ADVANCING.
121     ACCEPT CUS-NAME.
122     DISPLAY "ADDRESS OF CUSTOMER : "
123       WITH NO ADVANCING.
124     ACCEPT CUS-ADDR.
125     DISPLAY "FUNCTION OF CUSTOMER : "
126       WITH NO ADVANCING.
127     ACCEPT CUS-FUNCT.
128     DISPLAY "DATE : " WITH NO ADVANCING.
129     ACCEPT CUS-REC-DATE.
130
131 *E    EXEC SQL
132 *E     INSERT INTO CUSTOMER
133 *E        VALUES (:CUS-CODE, :CUS-NAME, :CUS-ADDR,
134 *E                :CUS-FUNCT, :CUS-REC-DATE)
135 *E    END-EXEC.
136     DIRECT-MAP CUS-CODE TO CUSTOMER--CODE.
137     DIRECT-MAP CUS-NAME TO CUSTOMER--NAME.
138     DIRECT-MAP CUS-ADDR TO CUSTOMER--ADDR.
139     DIRECT-MAP CUS-FUNCT TO CUSTOMER--FUNCT.
140    DIRECT-MAP CUS-REC-DATE TO CUSTOMER--REC-DATE.
141     INDIRECT-MAP CUSTOMER--CODE CUSTOMER--NAME
142        CUSTOMER--ADDR CUSTOMER--FUNCT
143        CUSTOMER--REC-DATE TO SLQCODE.
144
```

```
145     IF(SQLCODE = 0)
146 *E      EXEC SQL
147 *E        COMMIT
148 *E      END-EXEC
149     ELSE
150       DISPLAY "ERROR".
151
152  LIST-CUS.
153     DISPLAY "LISTS OF THE CUSTOMERS".
154 *E    EXEC SQL
155 *E      DECLARE ALL_CUST CURSOR FOR
156 *E      SELECT CODE, NAME, ADDR,
157                FUNCT, REC_DATE
158 *E      FROM CUSTOMER
159 *E      ORDER BY CODE
160 *E    END-EXEC.
161 *E    EXEC SQL
162 *E     OPEN ALL_CUST
163 *E    END-EXEC.
164     INDIRECT-MAP "" TO SQLCODE.
165     IF(SQLCODE = 0)
166       MOVE 0 TO END-FILE.
167     PERFORM READ-CUS UNTIL (END-FILE = 1).
168
169  READ-CUS.
170 *E    EXEC SQL
171 *E     FETCH ALL_CUST
172 *E     INTO :CUS-CODE, :CUS-NAME, :CUS-ADDR,
173 *E          :CUS-FUNCT, :CUS-REC-DATE
174 *E    END-EXEC.
175     INDIRECT-MAP "" TO CUSTOMER--CODE
176        CUSTOMER--NAME CUSTOMER--ADDR
177        CUSTOMER--FUNCT CUSTOMER--REC-DATE
178        SQLCODE.
179     DIRECT-MAP CUSTOMER--CODE TO CUS-CODE.
180     DIRECT-MAP CUSTOMER--NAME TO CUS-NAME.
181     DIRECT-MAP CUSTOMER--ADDR TO CUS-ADDR.
182     DIRECT-MAP CUSTOMER--FUNCT TO CUS-FUNCT.
183    DIRECT-MAP CUSTOMER--REC-DATE TO CUS-REC-DATE.
184
185     IF(SQLCODE = 0)
186       MOVE 1 TO END-FILE
187       DISPLAY CUS-CODE CUS-NAME
188       DISPLAY CUS-ADDR
189       DISPLAY CUS-FUNCT CUS-REC-DATE
190       PERFORM DISP-CUS-HISTORY.
191
192  DISP-CUS-HISTORY.
193 *E   EXEC SQL
194 *E     DECLARE CUS_HIST CURSOR FOR
195 *E     SELECT P.TOT, PR.NAME
196 *E     FROM PURCH P, PRODUCT PR
197 *E     WHERE P.PRODUCT = PR.CODE
198 E       AND P.CUSTOMER = :CUS-CODE
199 *E    END-EXEC.
200
201 *E    EXEC SQL
202 *E     OPEN CUS_HIST
203 *E    END-EXEC.
204     DIRECT-MAP CUS-CODE TO PURCH--CUSTOMER.
205     INDIRECT-MAP PURCH--CUSTOMER TO SQLCODE.
206
207     DISPLAY "PRODUCT    TOTAL".
208
209     PERFORM DISP-HISTORY UNTIL (SQLCODE NOT = 0).
210
211  DISP-HISTORY.
212 *E    EXEC SQL
213 *E     FETCH CUS_HIST
214 *E     INTO :PURCH-TOT, :PROD-NAME
215 *E    END-EXEC.
216     INDIRECT-MAP PURCH--CUSTOMER
```

```
217        TO PURCH--TOT PRODUCT--NAME SQLCODE.
218     DIRECT-MAP PURCH--TOT TO PURCH-TOT.
219     DIRECT-MAP PRODUCT--NAME TO PROD-NAME.
220     IF(SQLCODE = 0)
221       DISPLAY PROD-NAME PURCH-TOT.
222
223  NEW-PROD.
224     DISPLAY "NEW PRODUCT".
225     DISPLAY "PRODUCT NUMBER : "
226       WITH NO ADVANCING.
227     ACCEPT PROD-CODE.
228
229     DISPLAY "NAME : " WITH NO ADVANCING.
230     ACCEPT PROD-NAME.
231
232     DISPLAY "LEVEL : " WITH NO ADVANCING.
233     ACCEPT PROD-PRICE.
234
235 *E    EXEC SQL
236 *E     INSERT INTO PRODUCT
237 *E       VALUES (:PROD-CODE, :PROD-NAME,
238 *E               :PROD-PRICE)
239 *E    END-EXEC.
240     DIRECT-MAP PROD-CODE TO PRODUCT--CODE.
241     DIRECT-MAP PROD-NAME TO PRODUCT--NAME.
242     DIRECT-MAP PROD-PRICE TO PRODUCT--PROD-PRICE.
243     INDIRECT-MAP PRODUCT--CODE PRODUCT--NAME
244        PRODUCT--PROD-PRICE TO SQLCODE.
245     IF(SQLCODE = 0)
246 *E      EXEC SQL
247 *E        COMMIT
248 *E      END-EXEC
249     ELSE
250       DISPLAY "ERROR".
251
252  LIST-PROD.
253     DISPLAY "LIST OF PRODUCTS ".
254 *E    EXEC SQL
255 *E     DECLARE ALL_PROD CURSOR FOR
256 *E     SELECT CODE, NAME, PRICE
257 *E     FROM PRODUCT
258 *E     ORDER BY CODE
259 *E    END-EXEC.
260 *E    EXEC SQL
261 *E     OPEN ALL_PROD
262 *E    END-EXEC.
263     INDIRECT-MAP "" TO SQLCODE.
264     PERFORM READ-PROD UNTIL SQLCODE NOT = 0.
265
266  READ-PROD.
267 *E    EXEC SQL
268 *E     FETCH ALL_PROD
269 *E     INTO :PROD-CODE, :PROD-NAME, :PROD-PRICE
270 *E    END-EXEC.
271     INDIRECT-MAP "" TO PRODUCT--CODE PRODUCT--NAME
272        PRODUCT--PROD-PRICE SQLCODE.
273     DIRECT-MAP PRODUCT--CODE TO PROD-CODE.
274     DIRECT-MAP PRODUCT--NAME TO PROD-NAME.
275     DIRECT-MAP PRODUCT--PROD-PRICE TO PROD-PRICE.
276     IF(SQLCODE = 0)
277       DISPLAY PROD-CODE
278       DISPLAY PROD-NAME
279       DISPLAY PROD-PRICE.
280
281  NEW-ORD.
282     DISPLAY "NEW ORDER".
283     DISPLAY "ORDER NUMBER : "
284       WITH NO ADVANCING.
285     ACCEPT ORD-CODE.
286
287     MOVE 1 TO SQLCODE.
288     PERFORM READ-CUS-CODE UNTIL SQLCODE = 0.
```

```
289     DISPLAY CUS-NAME.
290 *E    EXEC SQL
291 *E     INSERT INTO ORDERS VALUES (:ORD-CODE,
292 *E       SYSDATE(), :CUS-CODE)
293 *E    END-EXEC.
294     DIRECT-MAP ORD-CODE TO ORDER--CODE.
295     DIRECT-MAP CUS-CODE TO ORDER--CUSTOMER.
296     INDIRECT-MAP ORDER--CODE ORDER--CUSTOMER
297        TO SQLCODE.
298     IF(SQLCODE NOT = 0)
299       DISPLAY "ERROR"
300     ELSE
301       MOVE 0 TO END-FILE
302       PERFORM READ-DETAIL
303         UNTIL END-FILE = 1
304
305 *E     EXEC SQL
306 *E       COMMIT
307 *E     END-EXEC.
308
309  READ-CUS-CODE.
310     DISPLAY "CUSTOMER NUMBER : "
311       WITH NO ADVANCING.
312     ACCEPT CUS-CODE.
313 *E    EXEC SQL
314 *E     SELECT CODE
315 *E     INTO :CUS-CODE
316 *E     FROM CUSTOMER
317 *E     WHERE CODE = :CUS-CODE
318 *E    END-EXEC.
319     DIRECT-MAP CUS-CODE TO CUSTOMER--CODE.
320     INDIRECT-MAP CUSTOMER--CODE
321        TO CUSTOMER--CODE SQLCODE.
322     DIRECT-MAP CUSTOMER--CODE TO CUS-CODE.
323
324     IF(SQLCODE NOT = 0)
325       DISPLAY "NO SUCH CUSTOMER".
326
327  READ-DETAIL.
328     DISPLAY "PRODUCT CODE (0 = END) : ".
329     ACCEPT PROD-CODE.
330     IF PROD-CODE = 0
331       MOVE 1 TO END-FILE
332     ELSE
333       PERFORM READ-PROD-CODE.
334
335  READ-PROD-CODE.
336 *E    EXEC SQL
337 *E     SELECT CODE INTO :PROD-CODE
338 *E     FROM PRODUCT
339 *E     WHERE CODE = :PROD-CODE
340 *E    END-EXEC.
341     DIRECT-MAP PROD-CODE TO PRODUCT--CODE.
342     INDIRECT-MAP PRODUCT--CODE
343        TO PRODUCT--CODE SQLCODE.
344     DIRECT-MAP PRODUCT--CODE TO PROD-CODE.
345     IF SQLCODE = 0
346       PERFORM UPDATE-ORD-DETAIL
347     ELSE
348       DISPLAY "NO SUCH PRODUCT".
349
350  UPDATE-ORD-DETAIL.
351     DISPLAY "QUANTITY ORDERED : "
352       WITH NO ADVANCING
353     ACCEPT DET-QTY.
354 *E    EXEC SQL
355 *E     SELECT ORD_QTY
356 *E     FROM DETAIL
357 *E     WHERE ORDER = :ORD-CODE
358 *E        AND PRODUCT = :PROD-CODE
359 *E    END-EXEC
360     DIRECT-MAP ORD-CODE TO DETAIL--ORDERS
```

```
361     DIRECT-MAP PROD-CODE TO DETAIL--PRODUCT
362     INDIRECT-MAP DETAIL--ORDERS DETAIL--PRODUCT
363        TO DETAIL--ORD-QTY SQLCODE.
364     IF SQLCODE = 0
365       DISPLAY "ERROR : ALREADY ORDERED"
366     ELSE
367 *E      EXEC SQL
368 *E       INSERT INTO DETAIL
369 *E         VALUES (:ORD-CODE, :PROD-CODE,:DET-QTY)
370 *E      END-EXEC
371       DIRECT-MAP ORD-CODE TO DETAIL--ORDERS.
372       DIRECT-MAP PROD-CODE TO DETAIL--PROD.
373       DIRECT-MAP DET-QTY TO DETAIL--ORD-QTY.
374       INDIRECT-MAP DETAIL--ORDERS DETAIL--PROD
375          DETAIL--ORD-QTY TO SQLCODE.
376       PERFORM UPDATE-CUS-HIST.
377
378  UPDATE-CUS-HIST.
379 *E    EXEC SQL
380 *E     SELECT TOT INTO :PURCH-TOT
381 *E     FROM PURCH
382 *E     WHERE CUSTOMER = :CUS-CODE
383 *E         AND PRODUCT = :PROD-CODE
384 *E    END-EXEC.
385     DIRECT-MAP CUS-CODE TO PURCH--CUSTOMER.
386     DIRECT-MAP PROD-CODE TO PURCH--PROD.
387     INDIRECT-MAP PURCH--CUSTOMER PURCH--PROD
388        TO PURCH--TOT SQLCODE.
389     DIRECT-MAP PURCH--TOT TO PURCH-TOT.
390     IF(SQLCODE = 0)
391 *E      EXEC SQL
392 *E       INSERT INTO PURCH VALUES(:CUS-CODE,
393 *E                         :PROD-CODE, :DET-QTY)
394 *E      END-EXEC
395       DIRECT-MAP CUS-CODE TO PURCH--CUSTOMER
396       DIRECT-MAP PROD-CODE TO PURCH--PROD
397       DIRECT-MAP DET-QTY TO PURCH--TOT
398       INDIRECT-MAP PURCH--CUSTOMER PURCH--PROD
399          PURCH--TOT TO SQLCODE
400     ELSE
401 *E      EXEC SQL
402 *E       UPDATE PURCH SET
403 *E           TOT = (:PURCH-TOT + :DET-QTY)
404 *E       WHERE CUSTOMER = :CUS-CODE
405 *E         AND PRODUCT = :PROD-CODE
406 *E      END-EXEC.
407       COMPUTE W-TMP = PURCH-TOT + DET-QTY
408       DIRECT-MAP W-TMP TO PURCH--TOT
409       DIRECT-MAP CUS-CODE TO PURCH--CUSTOMER
410       DIRECT-MAP PROD-CODE TO PURCH--PROD
411       INDIRECT-MAP PURCH--TOT PURCH--CUSTOMER
412          PURCH--PRODUCT TO SQLCODE.
413
414  LIST-ORD.
415     DISPLAY "LIST OF ORDERS ".
416 *E    EXEC SQL
417 *E     DECLARE ALL_ORD CURSOR FOR
418 *E     SELECT CODE, CUSTOMER
419 *E     FROM ORDERS
420 *E     ORDER BY CODE
421 *E    END-EXEC.
422 *E    EXEC SQL
423 *E     OPEN ALL_ORD
424 *E    END-EXEC.
425     INDIRECT-MAP "" TO SQLCODE
426     MOVE 0 TO END-FILE
427     PERFORM READ-ORD UNTIL END-FILE = 1.
428
429  READ-ORD.
430 *E    EXEC SQL
431 *E     FETCH ALL_ORD
432 *E     INTO :ORD-CODE, :ORD-CUSTOMER
```

```
*E    END-EXEC.
    INDIRECT-MAP "" TO ORDER--CODE
       ORDER--CUSTOMER SQLCODE.
    DIRECT-MAP ORDER--CODE TO ORD-CODE.
    DIRECT-MAP ORDER--CUSTOMER TO ORD-CUSTOMER.
    IF(SQLCODE = 0)
      MOVE 1 TO END-FILE
      DISPLAY "ORD-CODE "
        WITH NO ADVANCING
      DISPLAY ORD-CODE
      DISPLAY "ORD-CUSTOMER "
        WITH NO ADVANCING
      DISPLAY ORD-CUSTOMER
      DISPLAY "ORD-DETAIL "
*E      EXEC SQL
*E       DECLARE ORD_DEF CURSOR FOR
*E       SELECT D.ORD_QTY, P.NAME
*E       FROM DETAIL D, PRODUCT P
*E       WHERE D.ORDER = :ORD-CODE
*E         AND D.PRODUCT = P.CODE
*E      END-EXEC
*E      EXEC SQL
*E       OPEN ALL_ORD
*E      END-EXEC
      DIRECT-MAP ORD-CODE TO DETAIL--ORDERS
      INDIRECT-MAP DETAIL--ORDERS TO SQLCODE
      DISPLAY "PROD NAME    QUANTITY"
      PERFORM DISPLAY-DETAIL
        UNTIL SQLCODE NOT = 0.

 DISPLAY-DETAIL.
*E    EXEC SQL
*E     FETCH ORD_DEF
*E       INTO :DET-QTY, :PROD-NAME
*E    END-EXEC.
    INDIRECT-MAP DETAIL--ORDERS TO DETAIL--ORD-QTY
       PRODUCT--NAME SQLCODE
    DIRECT-MAP DETAIL--ORD-QTY TO DET-QTY
    DIRECT-MAP PRODUCT--NAME TO PROD-NAME
    DISPLAY PROD-NAME DET-QTY.
```

ANNEX C *Strange Data Structures / real case studies*

This annex describes physicals data structures found in real programs (case studies). One of the big question about these physical structures is how to represent them in the DB-MAIN CASE tool and how to transform them into conceptual structures. Many of these data structures do not match to classical (academic) data structures.

For each example, the source code is given with its corresponding physical schema and the physical schema is transformed to obtain a more conceptual schema.

C.1. Chained lists

Files: imppc.cob, accpc.cob, leccle.cob, lecper.cob

This example shows the implementation of a double chained list using a relative COBOL file.

C.1.1. COBOL

The declaration of the three COBOL files are given in figure 187. Only the declaration of the files is given and not the complete source code (more than 1400 LOC), some fragments of the code are presented to explain constraints discovery.

```
SELECT PERCLE ASSIGN TO "PERCLE.RAN"
ORGANIZATION IS RELATIVE
ACCESS MODE IS DYNAMIC
RELATIVE KEY IS RELKEY-PERCLE
FILE STATUS IS FSTATCU1.

SELECT CLES ASSIGN TO "CLES.RMS"
ORGANIZATION IS INDEXED
ACCESS MODE IS DYNAMIC
RECORD KEY IS NUMCLE-CLE
ALTERNATE RECORD KEY IS EMPLAC-CLE
ALTERNATE RECORD KEY IS SUITE-CLE
     WITH DUPLICATES
FILE STATUS IS FSTATCU1.

SELECT PERSO ASSIGN TO "PERSO.RMS"
ORGANIZATION IS INDEXED
ACCESS MODE IS DYNAMIC
RECORD KEY IS MATR-PER
ALTERNATE RECORD KEY IS ALTKEY1-PER
FILE STATUS IS FSTATCU1.
```

```
FD  PERCLE LABEL RECORD STANDARD.
01  ART-PERCLE
   02  MATR-PERCLE            PIC X(4).
   02  NUMCLE-PERCLE          PIC X(10).
   02  NOMBRE-PERCLE          PIC 9(4).
   02  POINT-SUIVANT-PER-PERCLEPIC 9(7).
   02  POINT-PRECEDENT-PER-PERCLE
                               PIC 9(7).
   02  POINT-SUIVANT-CLE-PERCLEPIC 9(7).
   02  POINT-PRECEDENT-CLE-PERCLE
                               PIC 9(7).
01  ART-1-PERCLE.
   02  FILLER                 PIC X(32).
   02  1ER-LIBRE-LOGIQUE-PERCLEPIC 9(7).
   02  1ER-LIBRE-PHYSIQUE-PERCLE
                               PIC 9(7).
FD  CLES LABEL RECORD STANDARD.
01  ART-CLES.
   02  NUMCLE-CLE             PIC X(10).
   02  EMPLAC-CLE             PIC X(9).
   02  POINT-ANCIEN-PERCLE-CLE PIC 9(7).
   02  POINT-RECENT-PERCLE-CLE PIC 9(7).

FD  PERSO LABEL RECORD STANDARD.
01  ART-PERSO.
   02  MATR-PER               PIC X(4).
   02  NOM-PER                PIC X(30).
   02  ALTKEY1-FICHIER        PIC X(10).
   02  POINT-ANCIEN-PERCLE-PER PIC 9(7).
   02  POINT-RECENT-PERCLE-PER PIC 9(7).
```

FIGURE 187. The data structure declaration.

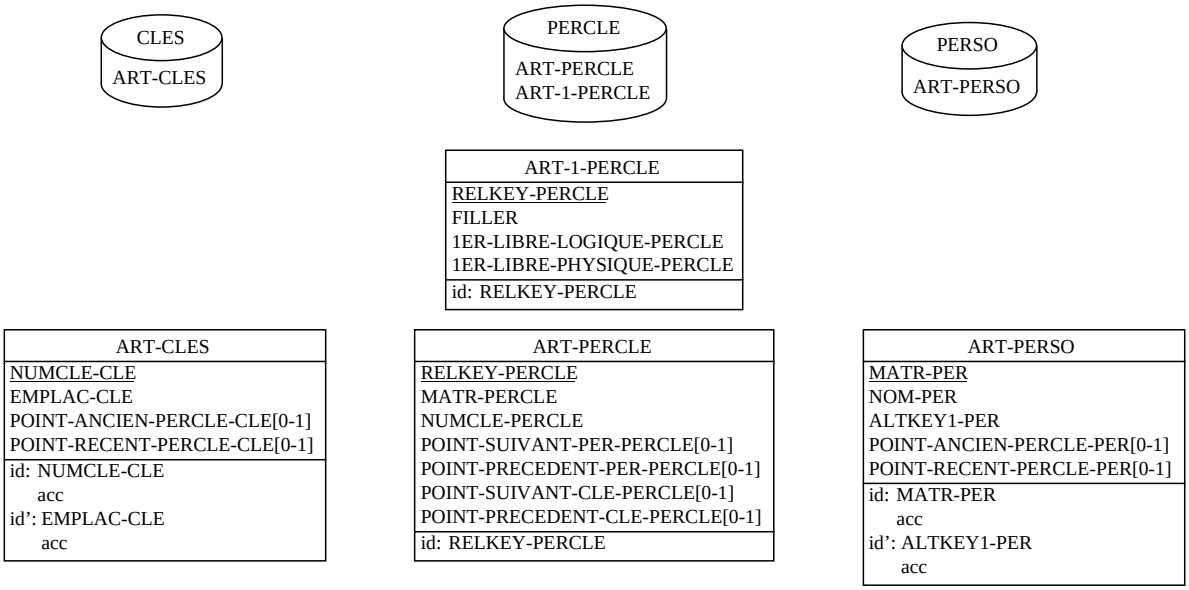


FIGURE 188. The raw physical schema.

C.1.2. The complete physical schema

PERCLE is a 'relative' file, so during the DDL code analysis, a technical attribute (RELKEY-PERCLE) is added to the entity types of PERCLE (ART-1-PERCLE and ART-PERCLE) to materialize the relative key. Each record can be access directly, giving the record number. RELKEY-PERCLE is the identifier of the entity type.

This example shows the implementation of two double chained lists between ART-CLES and ART-PERSO. These chained list are implemented by the two entity types of the collection PERCLE. A double chained list is a list that can be go through forward and backward. For example, the double chained list that has ART-CLES as origin is a list that from an occurrence of ART-CLES to all the occurrences of ART-PERSO that are connected can be reach sequentially from the first one of the list or starting from the last one.

The analysis of ART-1-PERCLE usage shows that the only occurrence of this entity types is the first record of PERCLE and all the other are to type ART-PERCLE. The attribute 1ER-LIBRE-LOGIQUE-PERCLE is the record number of the first unused (free) record of PERCLE, all the unused records are chained (using POINT-SUIVANT-PER-PERCLE). A referential constraint is created form 1ER-LIBRE-LOGIQUE-PERCLE to ART-PERCLE.RELKEY-PERCLE. 1ER-LIBRE-PHYSIQUE-PERCLE is the number of the record that follow the last record of the file. If there is no free record into the file then 1ER-LIBRE-LOGIQUE-WO and 1ER-LIBRE-PHYSIQUE-WO have the same value.

```
CREATION-LIEN.
```

Creates the double chained lists between ART-PERSO and ART-CLES (A)

```
IF POINT-RECENT-PERCLE-PER = 0
 MOVE 0 TO POINT-PRECEDENT-PER-PERCLE OF
     ART-PERCLE-WO
  MOVE 1ER-LIBRE-LOGIQUE-WO TO POINT-
     ANCIEN-PERCLE-PER
ELSE
 MOVE POINT-RECENT-PERCLE-PER TO
   POINT-PRECEDENT-PER-PERCLE
     OF ART-PERCLE-WO
   RELKEY-PERCLE
READ PERCLE
MOVE 1ER-LIBRE-LOGIQUE-WO TO
  POINT-SUIVANT-PER-PERCLE
    OF ART-PERCLE
REWRITE ART-PERCLE.
```

Update ART-CLE (B)

```
IF POINT-RECENT-PERCLE-CLE = 0
  MOVE 0 TO POINT-PRECEDENT-CLE-PERCLE
            OF ART-PERCLE-WO
  MOVE 1ER-LIBRE-LOGIQUE-WO TO
    POINT-ANCIEN-PERCLE-CLE
ELSE
  MOVE POINT-RECENT-PERCLE-CLE TO
    POINT-PRECEDENT-CLE-PERCLE
      OF ART-PERCLE-WO
    RELKEY-PERCLE
READ PERCLE
MOVE 1ER-LIBRE-LOGIQUE-WO TO
  POINT-SUIVANT-CLE-PERCLE
    OF ART-PERCLE
REWRITE ART-PERCLE.
```

Update the pointers to the last record into ART-PERSO and ART-CLE (C)

```
MOVE 1ER-LIBRE-LOGIQUE-WO TO
  POINT-RECENT-PERCLE-PER
  POINT-RECENT-PERCLE-CLE.
```

Fill ART-PERCLE-WO (D)

```
MOVE MATR-PER TO MATR-PERCLE
                  OF ART-PERCLE-WO.
MOVE NUMCLE-CLE TO NUMCLE-PERCLE
                  OF ART-PERCLE-WO.
MOVE 1 TO
  NOMBRE-PERCLE OF ART-PERCLE-WO.
MOVE 0 TO POINT-SUIVANT-PER-PERCLE
            OF ART-PERCLE-WO.
MOVE 0 TO POINT-SUIVANT-CLE-PERCLE
            OF ART-PERCLE-WO.
```

Storage of ART-PERCLE-WO (E)

```
MOVE 1ER-LIBRE-LOGIQUE-WO TO
  RELKEY-PERCLE.
MOVE CORRESPONDING ART-PERCLE-WO TO
  ART-PERCLE-LI.
IF 1ER-LIBRE-LOGIQUE-WO
            = 1ER-LIBRE-PHYSIQUE-WO
  MOVE CORRESPONDING ART-PERCLE-WO TO
    ART-PERCLE
  WRITE ART-PERCLE
  ADD 1 TO 1ER-LIBRE-LOGIQUE-WO
           1ER-LIBRE-PHYSIQUE-WO
ELSE
  READ PERCLE
  MOVE POINT-SUIVANT-PER-PERCLE
         OF ART-PERCLE TO
    1ER-LIBRE-LOGIQUE-WO
  MOVE CORRESPONDING ART-PERCLE-WO TO
    ART-PERCLE
  REWRITE ART-PERCLE.
```

Update of ART-PERCLE-1 (F)

```
MOVE SPACES TO ART-PERCLE.
MOVE 1ER-LIBRE-LOGIQUE-WO  TO
  1ER-LIBRE-LOGIQUE-PERCLE.
MOVE 1ER-LIBRE-PHYSIQUE-WO TO
  1ER-LIBRE-PHYSIQUE-PERCLE.
MOVE 1 TO RELKEY-PERCLE.
REWRITE ART-PERCLE-1.
```

FIGURE 189. Fragment of code that add a new record in PERCLE.

The code of figure 189 shows the fragment of code that is used to add a new element into the lists that connect ART-CLES with ART-PERSO. ART-CLES and ART-PERSO contain valid value for which all constraints validations have been done. These records will be stored after the double chained lists have been created.

The first part of the code (A) connects ART-PERSO to the list. IF POINT-RECENT-PERCLE-PER = 0, ART-PERSO is not yet connected to a list, it is the first time ART-PER is connected to ART-CLES (0 represent the null value). Otherwise, the last element of the list is read and the pointer to the next element of the list (the one that will be added) is updated. The second part of the code (B) updates the connection to ART-CLES.

The (C) code fragment update the pointer to the last element of the list (the one that will be added).

The (D) code fragment fills the new ART-PERCLE-WO record. MATR-PERCLE and NUMCLE-PERCLE contains reference to ART-PERSO and ART-CLES respectively. POINT-SUIVANT-PER-PERCLE and POINT-SUIVANT-CLE-PERCLE have the default value 0 that represent a null value.

In (E) fragment ART-PERCLE is stored. If no free record are available (1ER-LIBRE-LOGIQUE-PERCLE = 1ER-LIBRE-PHYSIQUE-PERCLE) the record is wrote at the end of the file, otherwise the first free record is rewrote.

The last fragment (F) updates the ART-PERCLE-1 record that contains the number of the first free record.

This code analysis can be summarized as follow.

Each element of ART-CLES is connected to a (optional) double chained list of elements of ART-PERCLE:

- POINT-ANCIEN-PERCLE-CLE references the first element of the optional list (this is why its cardinality is set to [0-1]).
- POINT-RECENT-PERCLE-CLE references the last element of the optional list (this is why its cardinality is set to [0-1]).
- POINT-SUIVANT-CLE-PERCLE points to the next element in the list, except for the last element of the list where it is equal to 0 (this is why its cardinality is set to [0-1]).
- POINT-PRECEDENT-CLE-PERCLE points to the previous element in the list, except for the first element of the list where it is equal to 0 (this is why its cardinality is set to [0-1]).

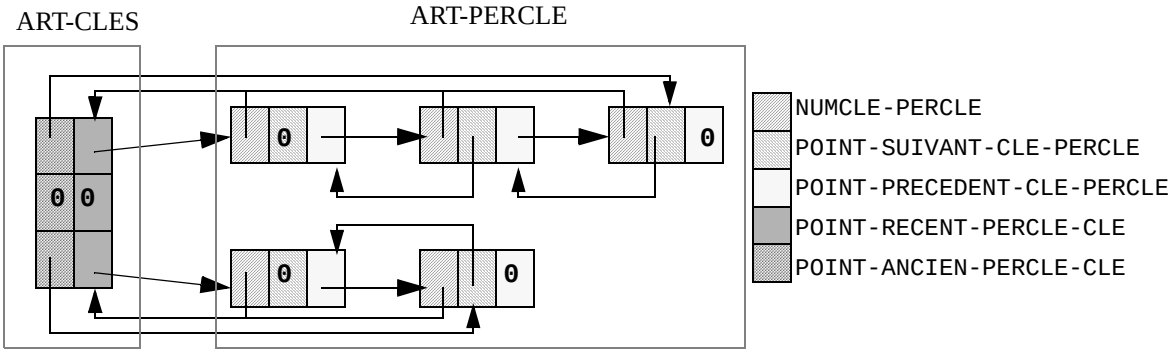


FIGURE 190. Example of the double chained list referenced by ART-CLES.

Figure 190 represents an example of such double chained list. Each record of ART-CLES is optionally connected to the first and last element of the list. Each record of ART-PERCLE has a pointer to the previous element of the list (except the first one), a pointer to the next element of the list (except the last one) and a pointer to the ART-CLES record.

There exist additional constraints about this chained list, to express some notations need to be defined:

$$\forall x \in \text{ART-CLES},\ \text{point-ancien-percle-cle}(x)$$
$$= \{y \in \text{ART-PERCLE} \mid x.\text{POINT-ANCIEN-PERCLE-CLE.POINT-SUIVANT-CLE-PERCLE..... } \rightarrow y$$
$$(\text{ between 0 and n time POINT-SUIVANT-CLE-PERCLE foreign key})\}$$

and

$$\forall x \in \text{ART-CLES},\ \text{point-recent-percle-cle}(x)$$
$$= \{y \in \text{ART-PERCLE} \mid x.\text{POINT-RECENT-PERCLE-CLE.POINT-PRECEDENT-CLE-PERCLE..... } \rightarrow y$$
$$(\text{ between 0 and n time POINT-PRECEDENT-CLE-PERCLE foreign key})\}$$

The additional constraints are:

$$(\forall x \in \text{ART-CLES})\text{point-ancien-percle-cle}(x) = \text{point-recent-percle-cle}(x)$$

and

$$(\forall y \in \text{ART-PERCLE}, \exists! x \in \text{ART-CLES}) \mid y \in \text{point-ancien-percle-cle}(x)$$

For the second list, each element of `ART-PERSO` can be connected to a double chained list of element of `ART-PERCLE`:

- `POINT-ANCIEN-PERCLE-PER` references the first element of the optional list (this is why its cardinality is set to [0-1]).
- `POINT-RECENT-PERCLE-PER` references the last element of the optional list (this is why its cardinality is set to [0-1]).
- `POINT-SUIVANT-PER-PERCLE` points to the next element in the list, except for the last element of the list where it is equal to 0 (this is why its cardinality is set to [0-1]).
- `POINT-PRECEDENT-PER-PERCLE` points to the previous element in the list, except for the first element of the list where it is equal to 0 (this is why its cardinality is set to [0-1]).

With the additional constraints:

$$(\forall x \in \text{ART-PERSO})\text{point-ancien-percle-per}(x) = \text{point-recent-percle-per}(x)$$

and

$$(\forall y \in \text{ART-PERCLE}, \exists! x \in \text{ART-PERSO}) \mid y \in \text{point-ancien-percle-per}(x)$$

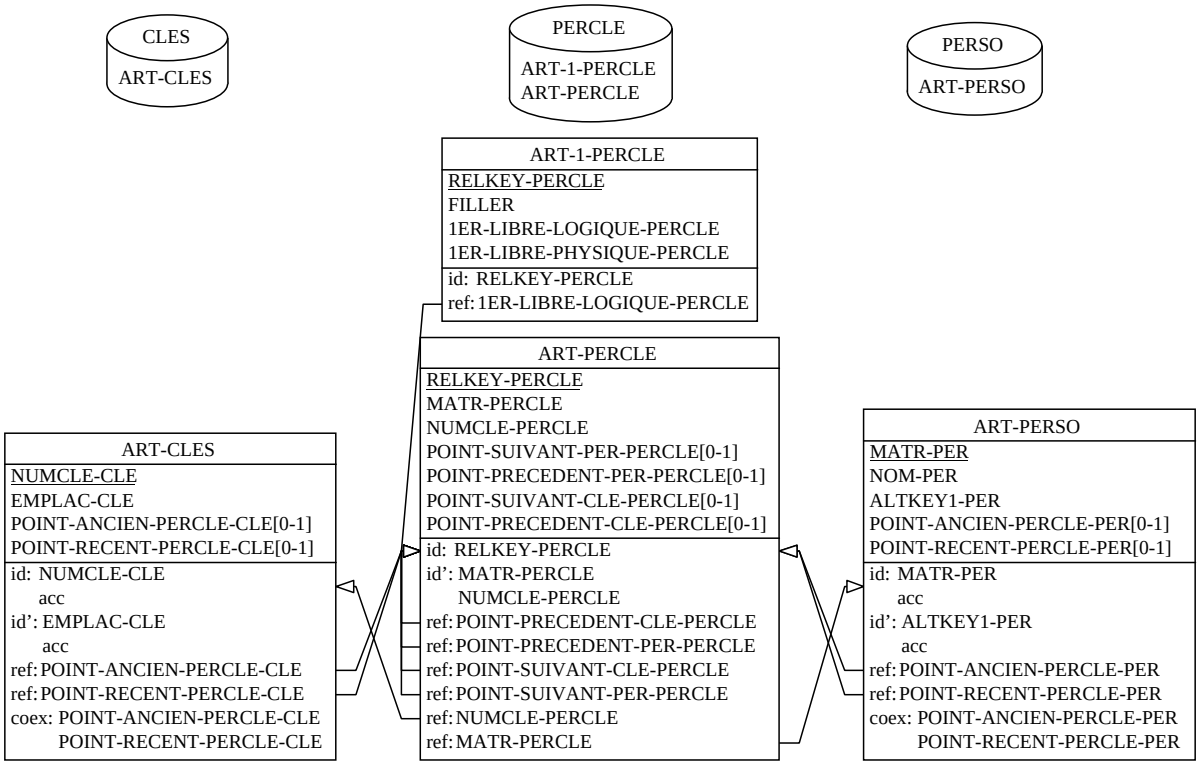


FIGURE 191. The complete logical schema.

C.1.3. Data structure conceptualization

C.1.3.1. Schema preparation

During the schema preparation, all the physical constructs can be removed and objects renamed:

- `ART-1-PERCLE` is an entity type that is only used to store some technical information, position of the first free record and the number of the last record of the file, thus it can be removed without any lost of semantic.
- The two double chained lists are only used to access the data and are redundant with the referential constraints that go from `ART-PERCLE` from `ART-PERCLE` to `ART-CLES` and to `ART-PERSO`. So the attributes used to represent those lists can be removed.
- `RELKEY-PERCLE` is a technical attribute that was added during the data structure extraction to materialize the record number and it has no semantic. It can be removed.
- The entity types can be renamed as the collections that have more meaningful names.
- The suffix of the attribute names can be removed, it is an usual COBOL program trick to prefix (in English) or to suffix (in French) attribute names to have unique name.
- The access keys and the collections can be removed.

The schema of figure 192 is the prepared schema.

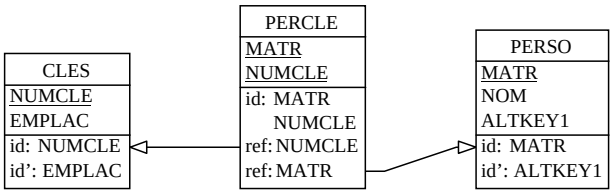


FIGURE 192. The prepared schema.

C.1.3.2. Untranslation and de-optimization

Obviously the two referential constraints can be transformed into relationship types.

C.1.3.3. Conceptualization

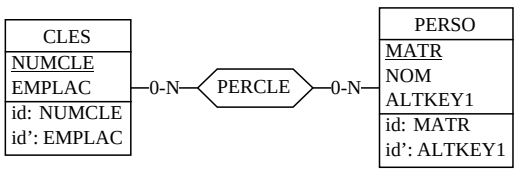


FIGURE 193. The conceptual schema.

The conceptualization of this small schema is simple, the only thing to do, is to transform the entity type `PERCLE` into a relationship type.

The final conceptual schema is displayed in figure 193.

C.2. Hierarchical foreign key

File: xclpc4.txt (carloc)

This example shows the usage of a hierarchical referential constraint in a COBOL/IDMS application. The IDMS database declaration and its corresponding physical schema are given in figure 194.

```
RECORD NAME IS XCLRC002
 LOCATION MODE IS VIA XCLSC001-C002 SET.
 02 C002-KEY3.
   03 C002-TYPE               PIC X(6).
   03 C002-KEY1               PIC X(10).
 02 C002-GARNITURE            PIC X(4).
RECORD NAME IS XCLRC004
 LOCATION MODE IS CALC USING (C004-KEY)
 DUPLICATES ARE NOT ALLOWED.
 02 C004-KEY                  PIC X(6).
RECORD NAME IS XCLRC007
 LOCATION MODE IS VIA XCLSC004-C007 SET.
 02 C007-KEY                  PIC X(5).
 02  C007-DESCR               PIC X(80).
 02  C007-ACTIF               PIC X.
SET NAME IS XCLSC004-C007
  OWNER IS XCLRC004
  MEMBER IS XCLRC007 LINKED TO OWNER
   MANDATORY AUTOMATIC
   KEY IS (C007-KEY ASCENDING)
     DUPLICATES ARE NOT ALLOWED.
```

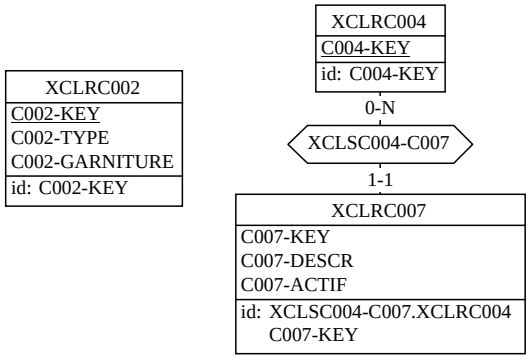


FIGURE 194. The database declaration and its raw physical schema.

C.2.1. Source code

```
OBTAIN CURRENT XCLRC002.
                                   Reads the current XCLRC002 record.
MOVE C002-TYPE TO C004-KEY.
OBTAIN CALC XCLRC004.
                                        Reads the XCLRC004 record.
IF DB-REC-NOT-FOUND
  ...
ELSE
  MOVE C002-GARNITURE TO C007-CAPITONNAGE
  OBTAIN XCLRC007 WITHIN XCLSC004-C007 USING C007-CAPITONNAGE
            Reads the XCLRC007 record that is connected to the current
                          XCLRC004 through the XCLSC004-C007 set.
  IF DB-REC-NOT-FOUND
    ...
  ELSE
    MOVE C007-DESCR TO MRC4-AUSST-DESCR
  END-IF.
END-IF.
```

FIGURE 195. The procedural fragment.

Figure 195 is a procedural fragment that implement the hierarchical referential constraint usage in COBOL/IDMS.

C.2.2. The complete logical schema

The analysis of the first three lines of the fragment show a trivial referential constraint between `C002-TYPE` and `XCLRC004`.

The analysis of the reminder of the code shows that another attribute of `XCLRC002` (`C002-GARNITURE`) is used to fill the record key of `XCLRC007` (`C007-KEY`), in fact this is not really the identifier, which is the role through `XCLRC004` and `C007-KEY`. But we have already find a referential constraint from `C002-TYPE` to `XCLRC004`, so we discovered a second referential constraint from (`C002-TYPE`, `C002-GARNITURE`) to the identifier of `XCLRC007`. This is a hierarchical referential constraint because the target of the referential constraint contains a role.

The logical schema is displayed in figure 196.

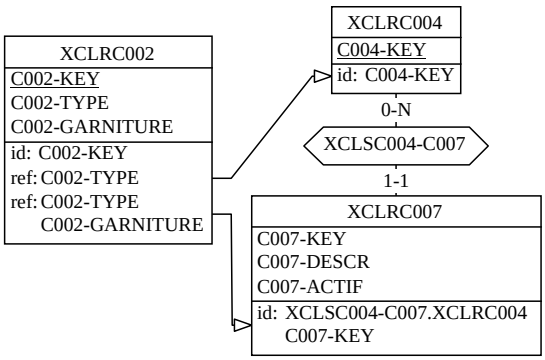


FIGURE 196. The complete logical schema.

C.2.3. Conceptualization

To conceptualize this schema, we have to transform the referential constraints. First of all we can notice that the first referential constraint (`C002-TYPE`, `C004-KEY`) is redundant with the second one, so it can be suppressed.

The second one can be transformed into a relationship type (*not accepted by DB-MAIN!*).

The conceptual schema is shown in figure 197.

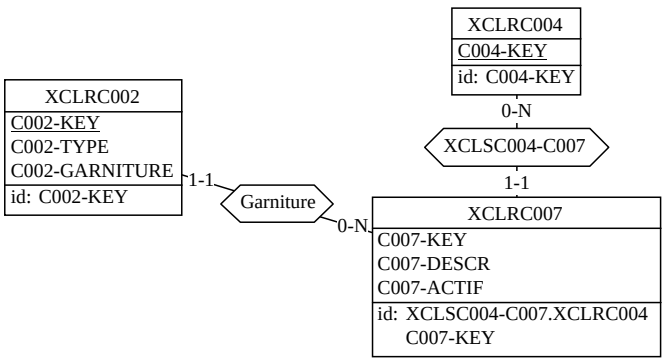


FIGURE 197. The conceptual schema.

C.3. Computed referential constraint (1)

File: XSNPS12.TXT

This example presents the implementation of a computed referential constraint. To find the referenced record, the value to the reference attribute is used as the input parameters of a function. In this example, the function is a translation table.

C.3.1. The DDL analysis

```
RECORD NAME IS XDDRD002
  LOCATION MODE IS VIA XDDSIX01-D002 SET.
02 D002-NR                    PIC X(6).
02 D002-USINE                 PIC X(15).
02 D002-DEALER                PIC X(04).
...
RECORD NAME IS XSNRS007
  LOCATION MODE IS VIA XSNSIX08-S007 SET.
02  S007-KEY.
  03  S007-P-L                PIC X.
  03  S007-DEALER             PIC X(04).
02  S007-DENOM                PIC X(15).
02  S007-TEL                  PIC X(15).
...
SET NAME IS XDDSIX01-D002
  OWNER IS SYSTEM
  MEMBER IS XDDRD002 ...KEY IS (D002-NR).
SET NAME IS XSNSIX08-S007
  OWNER IS SYSTEM
  MEMBER IS XSNRS007 MANDATORY AUTOMATIC
  KEY IS (S007-KEY ASCENDING)
  DUPLICATES ARE NOT ALLOWED.
```

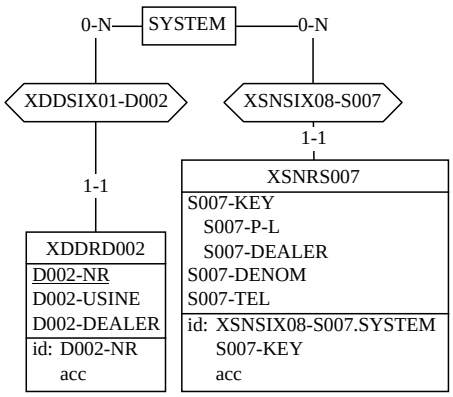


FIGURE 198. DDL declaration and the corresponding raw physical schema.

This example uses an IDMS database. The DDL and its corresponding raw physical schema are given in figure 198. IDMS databases are hierarchical database, i.e. the programmer can declare *set*, some kind of relationship type. In an IDMS database there exists a special entity type, named `SYSTEM`, to which correspond exactly one record. Its main usage is to be the *owner* of some sets to support sort or access keys. This trick is necessary because the only manner to declare sorted access key in IDMS is to declare them into a set. So if a record is the member of no sets and the programmer wants to declare an access key, he has to create a set that have this entity type as a member and `SYSTEM` as the owner and then to declare the access key on the set.

C.3.2. The schema refinement

```
...
OBTAIN XDDRD002 WITHIN XDDSIX06-D002 USING D002-NR.
IF DB-REC-NOT-FOUND THEN
  MOVE 'NOTFOUND' TO WR03-CURRENT
  GO TO END-PAR.
IF D002-USINE = 'TRUCK' THEN
  MOVE 'P' TO S007-P-L
ELSE
  MOVE 'L' TO S007-P-L.
MOVE D002-DEALER TO S007-DEALER.
OBTAIN XSNRS007 WITHIN XSNSIX08-S007 USING S007-KEY.
```

FIGURE 199. Procedural fragment.

The analysis of the procedural fragment (figure 199) shows that `D002-DEALER` is copied into `S007-DEALER` (part to the identifier) and that there exist a function (translation table) between `D002-USINE` and `S007-P-L` (the second part of the identifier). `S007-P-L` contains the value `P` or `L` depending of the value of `D002-USINE`. So this is a function that is used to compute the values of the identifier of `XSNRS007` according to the value of two attributes of `XDDRD002`. This is called a computed referential constraint.

Figure 200 represents the complete logical schema with the computed foreign key.

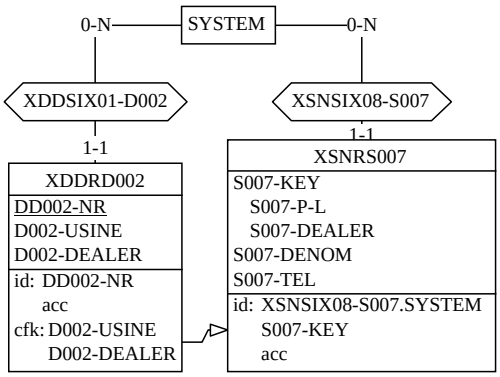


FIGURE 200. The complete logical schema.

C.3.3. The conceptualization

The first step of the conceptualization (preparation) is to remove the physical constructs. In this example, the entity type *SYSTEM* and its relationship types can be removed because they are only used to implement IDMS access key. The access keys can also be removed.

The computed referential constraint can be transformed into a relationship type, but `D002-USINE` could not be suppressed and a constraint has to be added to express the relation between `D002-USINE` and `S007-P-L` (see [Hainaut-1997a]).

The conceptual schema is shown in figure 201.

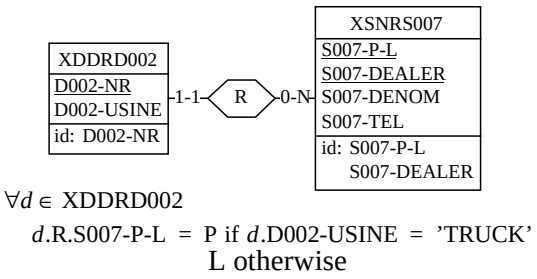


FIGURE 201. The conceptual schema.

C.4. Computed referential constraint (2) - Y2K

File: XDDPD8D.TXT

This is a typical year 2000 example, where a windowing solution has been implemented. This example use COBOL/IDMS.

C.4.1. The DDL analysis

```
RECORD NAME IS XDDRD022
  LOCATION MODE IS VIA XDDSD002-D022 SET.
...
02 D022-DELIVERY-DT.
  03 D022-DELIVERY-AA          PIC XX.
  03 D022-DELIVERY-MM          PIC XX.
  03 D022-DELIVERY-JJ          PIC XX.
RECORD NAME IS XDDRD010
  LOCATION MODE IS CALC USING (D010-KEY1)
  DUPLICATES ARE NOT ALLOWED.
02 D010-KEY1.
  03 D010-CE                   PIC XX.
  03 D010-AAMMJJ.
    05 D010-AA                 PIC XX.
    05 D010-MM                 PIC XX.
    05 D010-JJ                 PIC XX.
02 D010-TYPE                   PIC X.
```

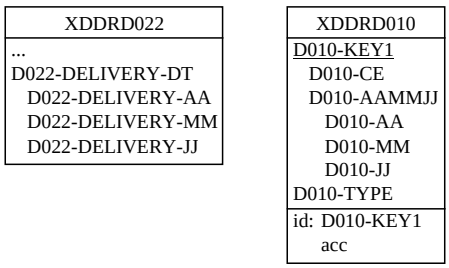


FIGURE 202. DDL declaration and its corresponding raw physical schema.

The DDL code and its corresponding raw physical schema are displayed in figure 202. It can be noticed in the physical schema that in `XDDRD022` the date contains only a year (`D022-DELIVERY-AA`) coded in two characters and the date in `XDDRD010` has a century attribute (`D010-CE`) of two characters and a year attribute (`D010-AA`) of two characters.

C.4.2. The schema refinement

```
...
OBTAIN FIRST XDDRD022
     WITHIN XDDSD002-D022.
MOVE D022-DELIVERY-AA TO D010-AA.
MOVE D022-DELIVERY-MM TO D010-MM.
MOVE D022-DELIVERY-JJ TO D010-JJ.
IF D010-AA > '90' THEN
  MOVE 19 TO D010-CE
ELSE
  MOVE 20 TO D010-CE.
OBTAIN XDDRD010 WITHIN XDDSIX13-D010
     USING D010-KEY1.
```

FIGURE 203. Procedural Fragment.

The analysis of the procedural code fragment (figure 203) shows that `D022-DELIVERY-AA` (`-MM`, `-JJ`) is copied into `D010-AA` (`-MM`, `-JJ`) and `D010-CE` is set to 19 or 20 according to the value of `D010-AA` (the implementation of the windowing algorithm with a cutoff year set to 1990 [IBM-1998]).

This can be represented into the logical schema (figure 204) by a computed referential constraint.

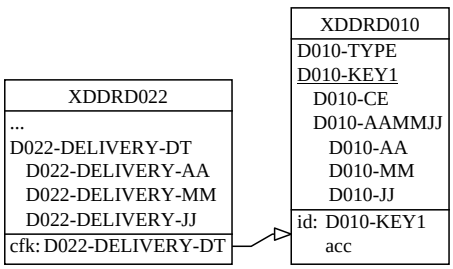


FIGURE 204. The complete logical schema.

C.4.2.1. The conceptualization

After the suppression of the access key (preparation), the computed referential constraint is transformed into a relationship type. A constraints is added to express that `XDDRD022` is connected to `XDDRD010` with a date comprise between 1st January 1991 and 31th December 2090.

The conceptual schema is shown in figure 205.

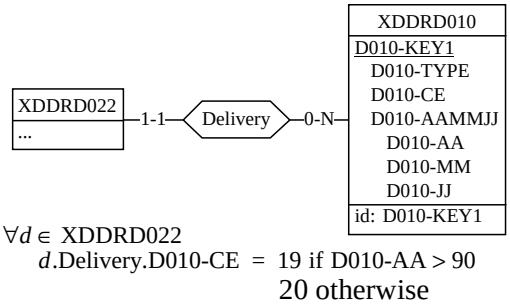


FIGURE 205. The conceptual schema.

C.5. Computed referential constraint (3)

File: XDDPD8D.TXT

This is an example of one of the simplest computed referential constraint, where the function concatenate a constant to the referential attribute.

C.5.1. The DDL analysis

```
RECORD NAME IS XDCRDC01
  LOCATION MODE IS VIA XDCSDC00-DC1A SET.
02 ...
02 DC01-USINE.
  03 DC01-USINE1              PIC X.
  03 DC01-USINE2-6            PIC X(06).
RECORD NAME IS XTLRT000
  lOCATION MODE IS CALC USING (T000-KEY)
  DUPLICATES ARE NOT ALLOWED;
02 T000-KEY.
  03 T000-TRUCK               PIC X(15).
  03 T000-USINE               PIC X(08).
02  T000-TXT                  PIC X(80).
```

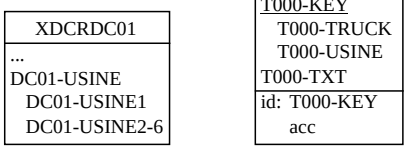


FIGURE 206. DDL declaration and its corresponding raw physical schema.

The DDL code and its corresponding raw physical schema are shown in figure 206.

C.5.2. The schema refinement

```
OBTAIN CURRENT XDCRDC01.
MOVE 'VEH-NEUFS' TO T000-TRUCK.
MOVE DC01-USINE2-6 TO T000-USINE.
OBTAIN CALC XTLRT000.
```

FIGURE 207. Procedural Fragment.

Through the analysis of the procedural fragment (figure 207), it can be noticed that `DC01-USINE2-6` is copied into `T000-USINE` and `T000-TRUCK` is set to a constant (’`VEH-NEUFS`’). `T000-TRUCK` and `T000-USINE` are the identifier, so this is a computed referential constraint.

The complete logical schema is shown in figure 208.

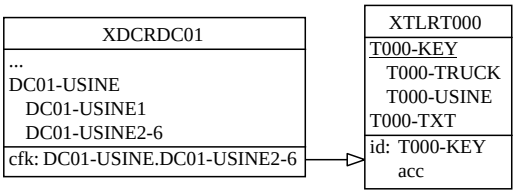
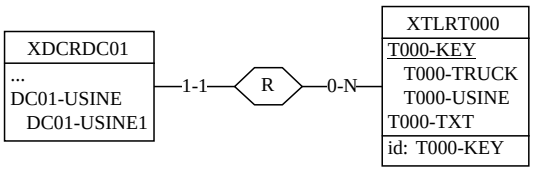


FIGURE 208. The complete logical schema.

C.5.3. The conceptualization

After the suppression of the access keys (preparation), the computed referential constraint is transformed into a relationship type. A constraint is added to express that all the `XTLRT00` connected to `XDCRDC01` through `R` must have `T000-TRUCK` = ’`VEH-NEUFS`’.

Figure 209 displays the conceptual schema.



$\forall d \in$ XDCRDC01
d.R.T000-TRUCK = ’VEH-NEUFS’

FIGURE 209. The conceptual schema.

C.6. Create a temporary file

File: budget04.cbl, line 841

In this example, a temporary file is created to sort a file, the record key of the temporary file is the sort criteria.

C.6.1. COBOL

Figure 210 shows the declaration of the files.

Figure 211 shows the fragment of the procedural code that copies the original file into the temporary one and use it to print a sorted report.

```
SELECT BUDTSEC  ASSIGN TO DISK, BUDTSECXX
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   FILE STATUS IS FILSTAT
   RECORD KEY IS BUDTSKEY.
SELECT BUDTREP ASSIGN TO DISK, BUDTREPXX
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   FILE STATUS IS FILSTAT
   RECORD KEY IS ANTRKEY.
SELECT BUDRPA ASSIGN TO DISK, BUDRPAXX
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   FILE STATUS IS FILSTAT
   RECORD KEY IS ORDRPAKEY.
```

```
FD BUDTSEC LABEL RECORD OMITTED.
* BUDTSEC TABLE DES SECTIONS EXISTANTES
01 BUDTSECREC.
  03 BUDTSKEY.
    05 BUDTSNR                PIC X(05).
  03 BUDTSLIB                 PIC X(30).
FD BUDTREP LABEL RECORD OMITTED.
* TABLE DE REPARTITION DES SECTIONS
01 ANTREPREC.
  03 ANTRKEY.
    05 ANTRNR                 PIC X(05).
  03 R-ANTRREST.
   05 ATRIND                  PIC X(04).
   05 ATRORD                  PIC 9(03).
FD BUDRPA LABEL RECORD OMITTED.
* ORDRE DE REPARTITION ANALYTIQUE
01 ORDRPAREC.
   03 ORDRPAKEY               PIC X(03).
   03 ORDRPANUM               PIC X(05).
```

FIGURE 210. The files and records declaration.

```
MJBUDRPA SECTION.
MJ-01.
  OPEN INPUT  BUDRPA.
  IF  FILSTAT  = "94"
    CLOSE BUDRPA
  ELSE
```

If the file BUDRPA exist, delete it.

```
    CLOSE BUDRPA
    DELETE FILE BUDRPA.
MJ-02.
  OPEN I-O BUDRPA.
  MOVE SPACES TO ANTRNR.
  START BUDTREP KEY > ANTRKEY
```

Positions on the first record.

```
    INVALID KEY GO TO MJ-EX.
MJ-03.
  READ BUDTREP NEXT AT END
    CLOSE BUDRPA GO TO MJ-05.
```

Reads the next record.

```
  MOVE ATRORD TO ORDRPAKEY.
  MOVE ANTRNR TO ORDRPANUM.
```

Fills ORDRPAREC.

```
  WRITE ORDRPAREC INVALID KEY
    GO TO MJ-04.
  GO  TO MJ-03.
MJ-04.
  CLOSE BUDRPA.
  GO TO MJ-EX.
MJ-05.
```

DUDRPA contains the fields ATRORD, ANTRNR sorted by ARTRORD while BUDTREP was ordered on ANTRNR.

```
  OPEN INPUT  BUDRPA.
  DISPLAY "ORDRE  DE REPARTITION ".
  DISPLAY "==================== ".
  MOVE SPACES TO ORDRPAKEY.
  START BUDRPA KEY > ORDRPAKEY
```

Select the first record.

```
    INVALID KEY GO TO MJ-EX.
MJ-06.
  READ BUDRPA NEXT AT END
```

Reads the next record.

```
    CLOSE BUDRPA GO TO MJ-EX.
  MOVE ORDRPANUM  TO BUDTSNR.
```

Fills the record key.

```
  READ BUDTSEC INVALID KEY
```

Reads the target record.

```
    MOVE SPACES TO BUDTSLIB.
  DISPLAY ORDRPAKEY, "  ", ORDRPANUM,
      " ", BUDTSLIB.
```

Prints an ordered report.

```
  GO  TO MJ-06.
MJ-EX.
  EXIT.
```

FIGURE 211. Procedural fragment.

C.6.2. The complete logical schema

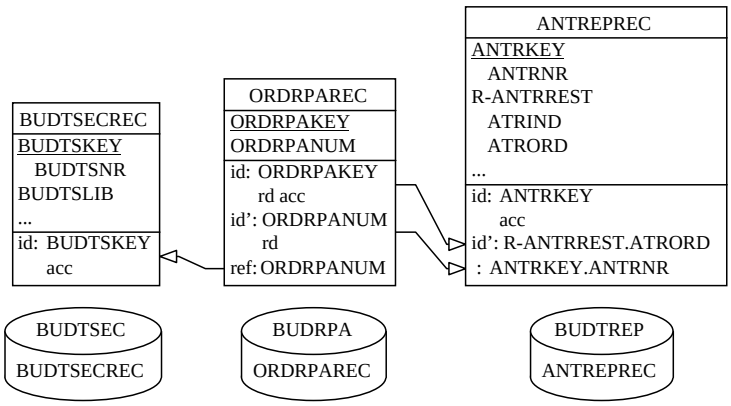


FIGURE 212. The complete logical schema.

The file BUDRPA is a temporary file, because at the beginning of the fragment it is deleted if it exists (figure 211, paragraph MJ-01).

Paragraph MJ-02 set the current record of BUDTREP to the first one. MJ-03 copies for each record of BUDTREP the value of ATRORD into ORDRPAKEY (the identifier) and ANTRNR into ORDRPANUM. ORDRPAREC is a copy of ANTREPREC, two rd constraints are added. ANTRKEY is an identifier, so its copy (ORDRPANUM) is also an (secondary) identifier. ORDRPAKEY is an identifier and a copy of ATRORD, so ATRORD is also an (secondary) identifier;

Through the analysis of paragraph MJ-06, it can be discovered that there is a referential constraint from ORDRPANUM to BUDTSKEY (BUDTSNR).

The complete logical schema is displayed in figure 212.

C.6.3. The conceptualization

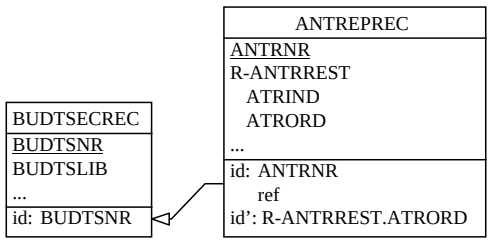


FIGURE 213. The prepared logical schema.

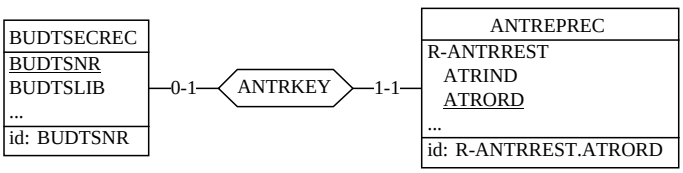


FIGURE 214. The conceptual schema.

The first step is to prepare the schema, i.e. remove all the unnecessary physical constructs. In this schema, all the access keys and the collections are removed. Some compound attributes (BUDTSKEY, ANTRKEY) have only one component, so they can be disaggregated.

ORDRPAREC is a copy of ANTREPREC, so it can be integrated into ANTREPREC. The result of this first step is shown in figure 213.

Now referential constraint is transformed into a relationship type. The referential constraint is between two identifiers, so the relationship type is a one-to-one relationship type. The conceptual schema is displayed in figure 214.

C.7. COBOL

```
SELECT SCTRAV     ASSIGN TO DISK, BEST5    03 SCT-REST.
  ORGANIZATION IS INDEXED                    05 SCTTAB.
  ACCESS MODE IS DYNAMIC                       07 SCTLAM OCCURS 102 TIMES
  FILE STATUS IS FILSTAT                                          PIC X(08).
  ALTERNATE RECORD KEY IS                    05 SCTETA            PIC X(01).
    SCT-BLO  WITH DUPLICATES                 05 SCTLON            PIC 9(03).
  RECORD KEY IS SCTKEYN.                     05 SCTLAR            PIC 9(03).
SELECT SCMLAM     ASSIGN TO DISK, BEST7      05 SCTHAU            PIC 9(03).
  ORGANIZATION IS INDEXED                    05 SCTTAG            PIC 9(02).
  ACCESS MODE IS DYNAMIC                     05 SCTTSC            PIC 9(02).
  FILE STATUS IS FILSTAT                     05 SCTMAJ            PIC 9(01).
  RECORD KEY IS SCMKEYN.
                                        FD SCMLAM LABEL RECORD OMITTED.
FD SCTRAV LABEL RECORD OMITTED.         01 SCMLAMREC.
01 SCTRAVREC.                              03 SCMKEYN.
  03 SCTKEYN.                               05 SCMARM            PIC 9(02).
    05 SCTARM              PIC 9(02).      05 SCMPOS            PIC 9(03).
    05 SCTCHA              PIC 9(04).      05 SCMSCI            PIC X(08).
    05 SCTBLO              PIC 9(04).    03 SCM-REST.
    05 SCTCLE REDEFINES SCTBLO PIC X(04).  05 SCMM2          PIC S9(9)V99 COMP.
                                          05 SCMHAU          PIC S9(9)V99 COMP.
```

FIGURE 215. The files declaration.

```
SC10-02.
                                        Read the next record
  READ SCTRAV NEXT AT END
    STOP RUN.
  IF SCTMAJ = 1
       If the record has already been integrated
                                        into the statistics, skip it
    GO TO SC10-02.
           Compute some intermediate values
  COMPUTE B = SCTTAG + 1.
  COMPUTE LH = SCTLON * SCTHAU.
  COMPUTE M2LAM ROUNDED = LH / 10000.
                                        Fill the record key
  MOVE SCTARM TO SCMARM.
  MOVE SCTLAM(B) TO SCMSCI.
  MOVE B TO SCMPOS.
                                    Access to the monthly record
  READ SCMLAM INVALID KEY
          If the record does not exist fill it with
                                                default value
    MOVE LOW-VALUES TO SCM-REST.
                                                Compute the aggregate
    ADD M2LAM TO SCMM2.
    COMPUTE SCMHAU = SCMHAU+(SCTHAU / 100).
    REWRITE SCMLAMREC INVALID KEY
                                        If the record does not exist
      GO TO LAME-11.
    GO TO LAME-02.
  LAME-11.
    WRITE SCMLAMREC INVALID KEY
      MOVE "INVALID WRITE SCUPAB" TO FOUT1
      MOVE SCUKEY TO FOUT2
      DISPLAY SCRFOUT
      STOP RUN.
  LAME-02.
        Marks the daily record as integrated into
                                        the monthly statistics
      MOVE 1 TO SCTMAJ.
      REWRITE SCTRAVREC INVALID KEY
        STOP RUN.
      GO TO SC10-02.
```

FIGURE 216. The procedural part.

Figure 215 shows the declarations of the two files. SCTRAV is the daily production and SCMLAM is the monthly production statistics.

Figure 216 shows the procedural fragment used to compute the monthly statistics from the daily production data.

C.7.1. The complete logical schema

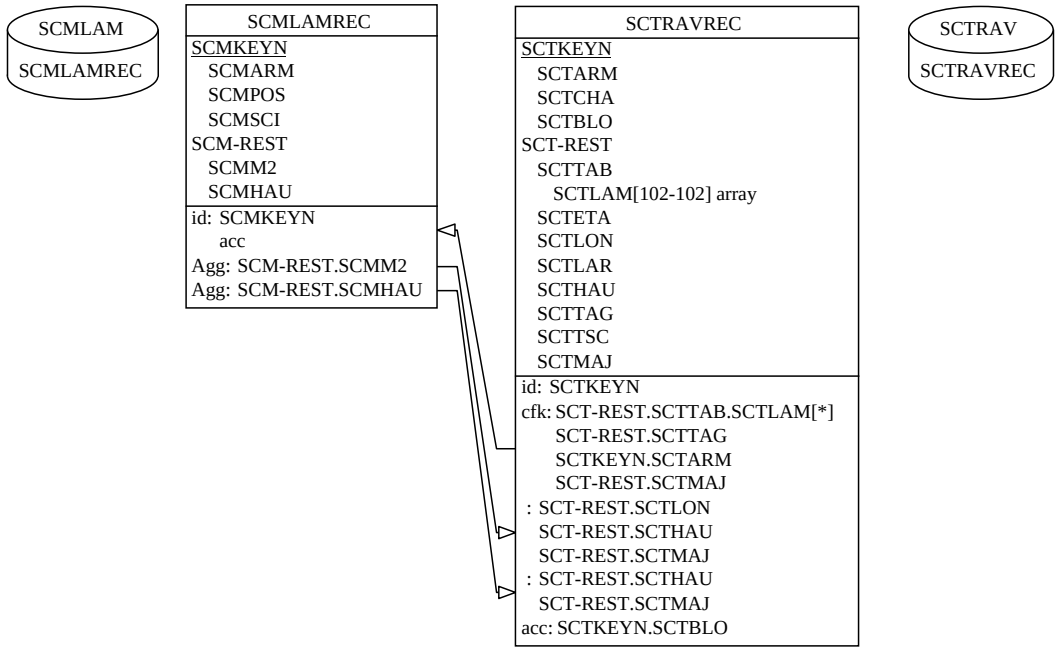


FIGURE 217. The complete logical schema.

The DDL code analysis is trivial, it produces two collections, two entity types with attributes, identifiers and access keys.

The procedural fragment is more difficult to interpret in spite of simplicity of the algorithm. The general algorithm is a big loop that reads each record of `SCTRAVREC` and performs the following actions:

- The value of the primary key of `SCMLAMREC` is computed.
- The corresponding record of `SCMLAMREC` is read or set to default value if it does not exist.
- The aggregate attributes of `SCMLAMREC` are updated.
- `SCMLAMREC` is written back to the file;
- The record `SCTRAVREC` is marked as read.

The tricky part of the fragment is that the record key of `SCMLAMREC` is not filled with the value of some attributes of `SCTRAVREC`, but a function of those attributes. `SCTTAG` is used as an index to access an element of `SCTLAM`. This is represented by a computed referential constraint that is only true if `SCTMAJ = 1`. The computed referential constraint can be expressed as follow:

- `SCMARM = SCTARM`
- `SCMPOS = SCTTAG + 1`
- `SCMSCI = SCTLAM(SCTTAG + 1)`

The values of `SCMM2` and `SCMHAU` are an aggregation of the values of the attribute of `SCTRAVREC`. These aggregations can be seen as business rules. These rules are stored in the description of the groups and can be expressed as follow:

- `SCMM2 = sum(SCTLON * SCTHAU / 10000)`

- `SCMHAU = sum(SCTHAU / 100)`

The complete logical schema is displayed in figure 217.

C.7.2. Conceptualization

The conceptualization of the schema is very easy, because `SCMLAMREC` is only the aggregation of the value of `SCTRAVRE`. So in the conceptual schema, `SCMLAMREC` can be removed. `SCTMAJ` can also be removed because it is only used to mark the records that have already been aggregate to `SCMLAMREC`.

C.8. History

File: b15.cob, line 2673, c10.cob

In this example, the file `CREREK` contains for each invoice the different payments. This file is updated with the records of a file (`HULP`) that contains the different financial operations.

C.8.1. COBOL

```
SELECT HULP ASSIGN TO DISK, BEST8
  ORGANIZATION IS INDEXED
  ACCESS  MODE IS DYNAMIC
  FILE STATUS IS  FILSTAT
  RECORD  KEY IS HULPKEY.
SELECT CREREK ASSIGN TO DISK, BEST1
  ORGANIZATION IS INDEXED
  ACCESS  MODE IS DYNAMIC
  RECORD  KEY IS CREREKKEY.

FD HULP LABEL RECORD OMITTED.
01 HULPREC.
  03 HULPKEY.
    05 HULPORG                PIC 9(4).
    05 HULPDAT.
      07 HULPDAA              PIC 99.
      07 HULPDMM              PIC 99.
      07 HULPDJJ              PIC 99.
    05 HULPIEC                PIC 9(5).
  03 HULPSCHERM.
    05 HULPDC                 PIC X.
    05 HULPREKALG.
      07 HGEN6OU7             PIC 999.
      07 HGENRES              PIC X(06).
    05 HULPREKAN.
      07 HANALCF              PIC X(05).
      07 HANALRES             PIC X(04).
    05 HULPDOK                PIC 9(5).
    05 HULPOMS                PIC X(25).
  03 HULPRDAT.
    05 HULPRDAA               PIC 99.
    05 HULPRDMM               PIC 99.
    05 HULPRDJJ               PIC 99.
  03 HULPBF             PIC S9(11) COMP.
  03 HULPSYMB                 PIC X.
FD CREREK
 LABEL RECORD OMITTED.
01 CREREC.
  03 CREREKKEY.
    05 CRKLANT                PIC X(5).
    05 CRDOKNR                PIC X(5).
    05 CRVOLG                 PIC 99.
  03 CRPER.
    05 CRJAAR                 PIC 99.
    05 CRMAAND                PIC 99.
  03 CRDATDOK.
    05 CRDATJAAR              PIC 99.
    05 CRDATMAAND             PIC 99.
    05 CRDATDAG               PIC 99.
  03 CROMSDOK                 PIC X(25).
  03 CRVERVD.
    05 CRVERVJAAR             PIC 99.
    05 CRVERVMAAND            PIC 99.
    05 CRVERVDAG              PIC 99.
  03 CRDEBCRE                 PIC X.
  03 CRBEDR                PIC S9(9)V99.
  03 CRAANZ                   PIC X.
  03 CRDATBET.
    05 CRDATBETJAAR           PIC 99.
    05 CRDATBETMAAND          PIC 99.
    05 CRDATBETDAG            PIC 99.
  03 CRSYMBP                  PIC X.
  03 CRBETAALD       PIC S9(9)V99 value
     0.
  03 CESCOMPTE       PIC S9(9)V99 value
     0.
  03 CFC                      PIC X.
  03 CRPIECP                  PIC 9(5).
```

FIGURE 218. Files and records declaration.

```
ACCEPT CRDATDAG.
ACCEPT CRDATMAAND.
ACCEPT CRDATJAAR.
ACCEPT CRKLANT.
ACCEPT CRBEDR.
MOVE NRNU TO CRDOKNR.
...
MOVE 0 TO CRVOLG.
MOVE 0 TO CRBETAALD CRBETAALD-EUR.
WRITE CREREC INVALID KEY
  MOVE "INVALID WRITE CREREK" TO FT1
  MOVE CREREKKEY TO FT2
  GO TO STOPS.
IF CESCOMPTE NOT =  0
  MOVE CESCOMPTE TO CRBETAALD
  MOVE CRDATDOK TO CRDATBET
  MOVE "G" TO CRSYMBP
  MOVE CRDOKNR TO CRPIECP
  REWRITE CREREC INVALID KEY
    GO TO STOPS.
```

FIGURE 219. Procedural code (creation of the customer in credit).

```
  MOVE BHULPIEC TO HULPIEC.
  MOVE BHULPORG TO HULPORG.
  MOVE BHULPDAT TO HULPDAT.
  START HULP KEY >= HULPKEY INVALID KEY
    GO TO I13.
I11.
  READ HULP NEXT AT END
    GO TO I13.
  IF HGEN6OU7 = 440
    PERFORM UPIMPFRS THRU FINIMPFRS.
  GO TO I11
UPIMPFRS.
*UPD FICHIER CREREK:IMPAYES CREDITEURS.
  MOVE HANALCF TO CRKLANT.
  MOVE HULPDOK TO CRDOKNR.
  MOVE 0 TO CRVOLG.
  IF HULPDOK = 0
    GO TO CREINC.
  START CREREK KEY=CREREKKEY INVALID KEY
    DISPLAY "INVALID START CREREK"
    STOP RUN.
READCREREK.
  READ CREREK NEXT AT END
    GOTO UPCREREK.
  MOVE CRVOLG TO BCRVOLG.
  IF CRBETAALD > 0
    GO TO DEJAPAI.
  MOVE HULPBF TO CRBETAALD.
  MOVE HULPSYMB TO CRSYMBP.
  MOVE HULPDAT TO CRDATBET.
  MOVE HULPIEC TO CRPIECP.
  IF HULPDC = "C" AND CFC = "0"
    MULTIPLY -1 BY CRBETAALD.
  IF HULPDC = "D" AND CFC = "1"
    MULTIPLY -1 BY CRBETAALD.
  COMPUTE SOLDE = CRBEDR - CRBETAALD.
  IF SOLDE = 0
    MOVE "A" TO CRAANZ.
  REWRITE CREREC INVALID KEY
    DISPLAY "INVALID REWRITE CREREK"
    GO TO STOPS.
  GO TO FINIMPFRS.
ECRICREREC.
  IF HULPDC = "D"
    MOVE HULPDAT TO CRDATBET
    MOVE HULPBF  TO CRBETAALD
    MOVE HULPSYMB TO CRSYMBP
    MOVE HULPIEC  TO CRPIECP
    MOVE 0 TO CRBEDR
  ELSE
    MOVE HULPBF  TO CRBEDR
    MOVE 0       TO CRBETAALD
    MOVE SPACES  TO CRSYMBP
    MOVE 0       TO CRPIECP.
  MOVE HULPDAA TO CRJAAR.
  MOVE PP TO CRMAAND.
  MOVE HULPDAT TO CRDATDOK CRVERVD.
  MOVE HULPOMS TO CROMSDOK.
  MOVE HULPDC TO CRDEBCRE.
  MOVE 0 TO CESCOMPTE.
  IF CRDOKNR=ZEROE
    MOVE SPACE TO CFC.
  WRITE CREREC INVALID KEY
    DISPLAY "INVALID WRITE CREREK!"
    STOP RUN.
  IF CRAANZ = "A"
    PERFORM SOLDEFR THRU FINSOLDEFRS.
  GO TO FINIMPFRS.
DEJAPAI.
  MOVE CRBEDR TO TDOC
  SUBTRACT CRBETAALD FROM TDOC.
  IF HULPBF = TDOC
    ADD 1 TO CRVOLG
    MOVE "A" TO CRAANZ
    GO TO ECRICREREC.
LECIMPC.
  READ CREREK NEXT AT END
    GOTO UPCREREK.
  IF CRKLANT NOT= HANALCF
    GOTO UPCREREK.
  IF CRDOKNR NOT= HULPDOK
    GOTO UPCREREK.
  MOVE CRVOLG TO BCRVOLG.
  SUBTRACT CRBETAALD FROM TDOC.
  IF TDOC = HULPBF
    ADD 1 TO CRVOLG
    MOVE "A" TO CRAANZ
    GO TO ECRICREREC.
  GO TO LECIMPC.
UPCREREK.
  MOVE HANALCF TO CRKLANT.
  MOVE HULPDOK TO CRDOKNR.
  COMPUTE CRVOLG = BCRVOLG + 1.
  IF HULPBF = TDOC
    MOVE "A" TO CRAANZ.
  GO TO ECRICREREC.
FINIMPFRS.
  EXIT.
```

FIGURE 220. The procedural code.

For this example, in addition the usual files and records declaration (figure 218), there is two procedural fragments. The first one (figure 219) is the creation of an entry for an invoice (new invoice), all the data are given interactively by the user. The second fragment (figure 220) reads, from a file, the different payment carried out and adds the corresponding records to the file.

This is also an example of a multi language development, some comments are in French (IMPAYES CREDITEURS = customer in credit unpaid), the variable names are mostly in Dutch (CRKLANT = CrCustomer or CRDOKNR = CrDokumentNumber = CrDocumentNumber) and some error messages are in English (INVALID START CREREK). This is a typical example of the Belgian way of doing, so the analyst needs to have at least some notion of the three languages and during the conceptualization all the names need to be translate in the same language.

C.8.2. The complete logical schema

The first fragment (figure 219) does not tell us any information about potential constraints, except that it tests the value of CESCOMPTE and if its value is different than 0, it is copied into CRBETAALD.

The second one (figure 220) is a complex fragment to analyze. In paragraph I11, this fragment only uses the record for which HGEN6OU7 = 440. The record key is constructed from two attributes of HULPREC (HANALCF and HULPDOK) and a constant (0).

In paragraph READCREREK, the record with the computed record key is read, if it does not exist (AT END) then a new record is added (paragraph UPCREREK). If the record exists and nothing has yet been paid (CRBETAALD = 0) the record is updated according to the value of HULPREC. If everything is paid (CRBEDR - CRBETAALD = 0) then CRAANZ is set to 'A'.

If something has already been paid (CRBETAALD > 0) then DEJAPAI is executed. In paragraph DEJAPAI, TDOC receives the value of CREBEDR (the amount of the invoice) and CRBETAALDT (amount already paid) is subtracted. So TDOC contains the reminder of the invoice to be paid. If TDOC = HULPBF (the amount paid) then one is added to CRVOLG, CRAANZ is set to 'A' and the paragraph ECRICREREC is executed (a record is written into the file). Otherwise, the next record is read (LECIMPC) until the end of the file is reached, or the next record does not belong to the same customer (CRKLANT) or does not belong to the same document (CRDOKNR), then UPCREREK is executed. For each new record, CRBETAALD is subtracted from TDOC, but CRBEDR is not used any more. So we can state that CRBEDR is optional and is only used for the first record of a couple CRKLANT, CRDOKNR.

Paragraph UPCREREK fills CRKLANT and CRDOKNR with the current customer number and document number and CRVOLD with the previous valued (BCRVOLG) plus one and then it executes ECRICREREC.

In paragraph ECRICREREC, the new record is filled with the value of HULPREC and the default values and it is written into the file. This can suggest that some attributes are optional (CRBEDR, CRBETAALD, CRSYMBP, CRPIECP, CESCOMPTE). If we look closer, we can notice that CRBETALD, CRSYMBP and CRPIECP all have a value or none of them have a value, so we add a coexistence constraints between them.

CRKLANT	CRDOKNR	CRVOLD	CRDEBCRE	CRBEDR	CRBETAALD	CRAANZ
K_1	D_1	0	C	bedr (not 0)	bet_0	aanz_0
K_1	D_1	1	D	0	bet_1	aanz_1
...	...	...	...	...	...	...
K_1	D_1	n	D	0	bet_n	aanz_n
K_2	D_2					

FIGURE 221. Example of the value of CREREK.

Figure 221 shows an example of the value of CREREK. We can express the relation between CRBEDR, CRBETAALD and CRAANZ for the records with the same CRKLNAT, CRDOKNR as follow:

if (bedr - (bet_0 + ... + bet_n)) = 0 then aanz_n = "A" and aanz_i = " " $\forall(0 \le i < n)$.
otherwise aanz_i = " " $\forall(0 \le i \le n)$.

It is very difficult (or impossible) to formalize this in the logical schema, we have discovered a business rule that express how CREREK is constructed and constraints about the structures of the data.

We define $\text{BALANCE}|_{k,d}$ as follow[1]:

$\forall k \in \text{CRKLANT}$, $\forall d \in \text{CRDOKNR}$ and $n = max(v | \exists \text{a record identified by}(k, d, v))$

$$\text{BALANCE}|_{k,d} = \text{CRDEBCRE}|_{k,d,0} - \sum_{0 \le i \le n} \text{CRBETAALD}|_{k,d,v}$$

The complete logical schema is the one displayed in figure 222 with the following constraints:

if $\text{BALANCE}|_{k,d} = 0$ then $\text{CRAANZ}|_{kdn} = ''\text{A}''$ and $\text{CRAANZ}|_{kdi} = '' '' \ \forall 0 \le i < n$
otherwise $\text{CRAANZ}|_{kdi} = '' '' \ \forall 0 \le i \le n$

$CRDEBCRE|_{k,d,0} \neq 0$ and $CRDEBCRE|_{k,d,v} = 0 \ \forall v < 0$

$CESCOMPTE|_{k,d,v} = 0 \ \forall 0 > v$

if $CESCOMPTE|_{k,d,0} \neq 0$ then $\text{CRBETAALD}|_{k,d,0} = \text{CESCOMPTE}|_{k,d,0}$

For all record of HULPREC with HGEN6OU7 = 440 then it exists *v* such that
$\text{CREBETTALD}|_{\text{HANALCF,HULPDOK, v}} = \text{HULPBF}$ and $\text{CRDATBET}|_{\text{HANALCF,HULPDOK, v}} = \text{HULPDAT}$ and
...

1. $ATT|_{k,d,v}$ =the value of the attribute ATT of entity type CREREK is identified by CRKLANT=*k*, CRDOKNR=*d* and CRVOLD=*v*.

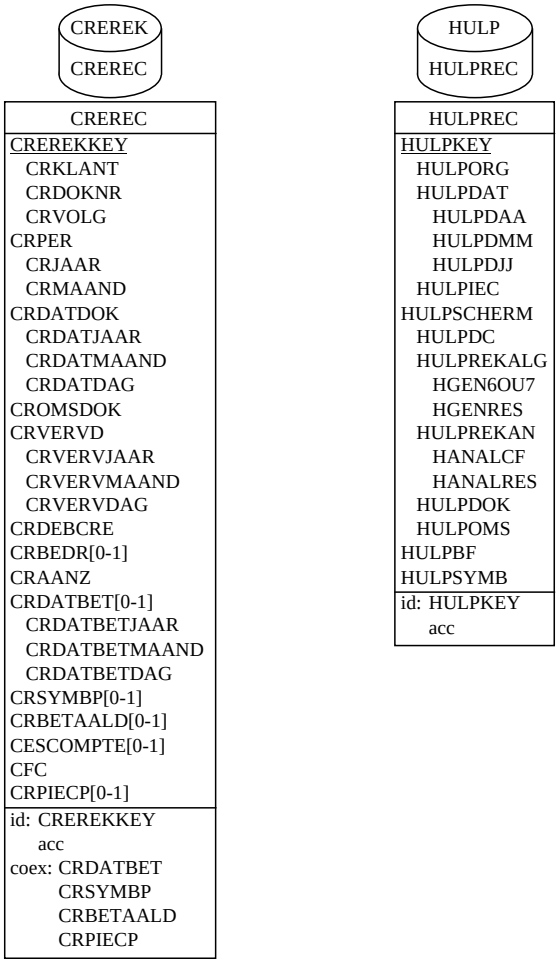


FIGURE 222. The complete logical schema.

C.8.3. Conceptualization

The conceptualization of this schema is not trivial because all the constraints are not expressed in the logical schema and some of those constraints are business rules. So the conceptualization is more an interpretation of the constraints than their transformation.

Our proposed solution is to divided the `CREREC` record into two entity types, one to represent the initial invoice, `CREREC`; and the otherone the different payments, `VOLG` (see figure 223). The attribute `CRAANZ` can be suppressed because it can be derived, it is used to mark when the entire invoice has been paid. If `CESCOMPTE` is different from 0, we did not copy it into `CRBETAALD`.

We can create a sub-type of `HULPREC` to materialize the entity type that have `HGEN6OU7` equal to 440 (see figure 223). We can create a referential constraint from (`HANALCF`, `HULPDOK`) to (`CRKLANT`, `CRDOKNR`).

The entity type `VOLG` is redundant with the entity type `HGEN6OU7=440`, so we can suppress it.

The referential constraint can be transformed into a relationship type. The final conceptual schema is shown in figure 224.

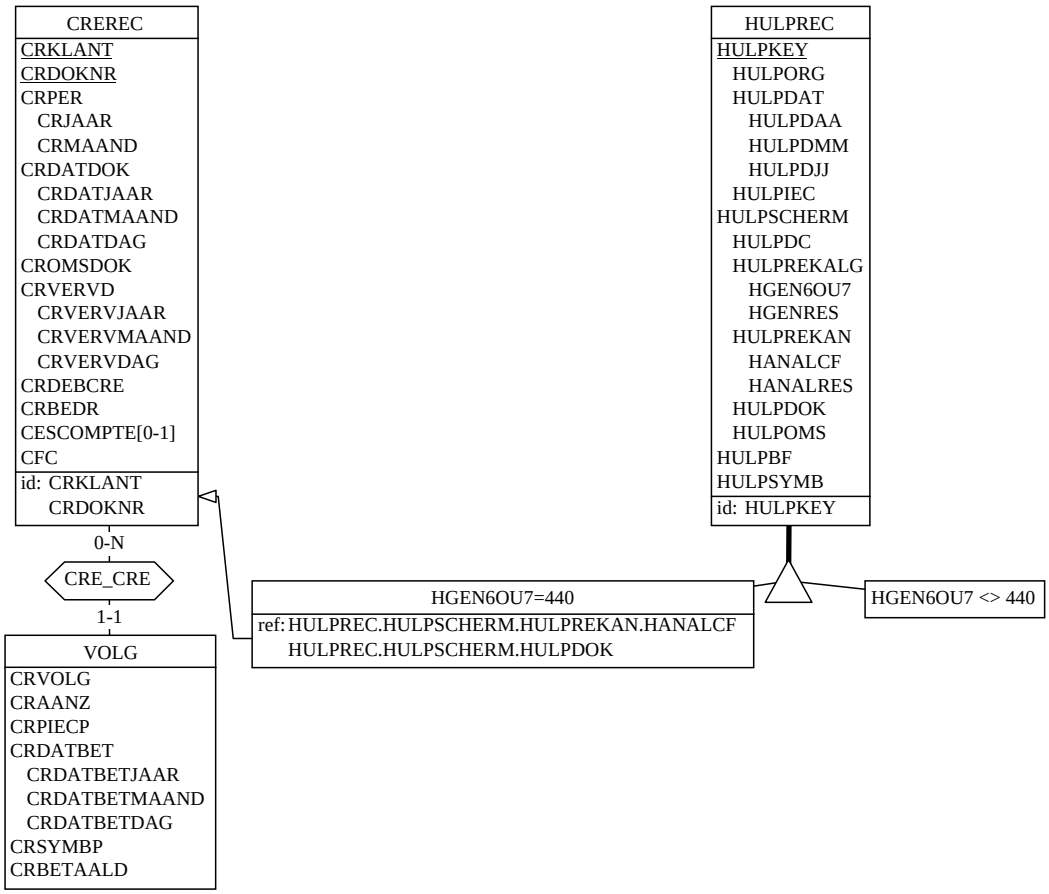


FIGURE 223. The raw conceptual schema.

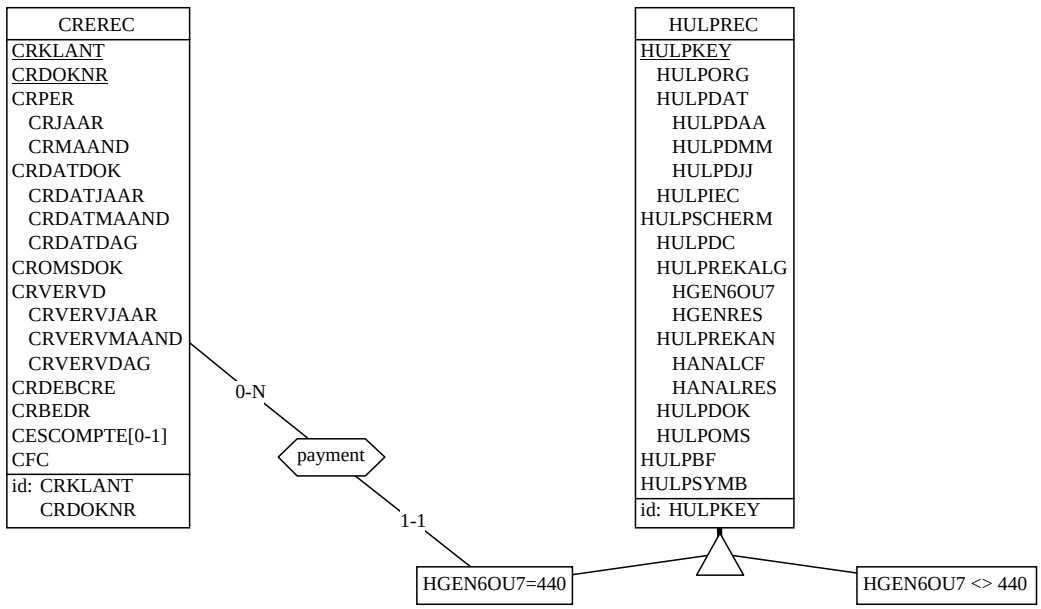


FIGURE 224. The conceptual schema.

C.9. Technical file

File: c10.cob

In this example, one of the files contains technical information. Data that are not part of the domain of the application. In this example, the technical file contains the last number assigned to an identifier.

C.9.1. COBOL

```
SELECT TABF ASSIGN TO DISK, BEST6
  ORGANIZATION IS INDEXED
  ACCESS  MODE IS DYNAMIC
  FILE STATUS IS  FILSTAT
  RECORD  KEY IS TABFKEY.
SELECT CREREK ASSIGN TO DISK, BEST1
  ORGANIZATION IS INDEXED
  ACCESS  MODE IS DYNAMIC
  RECORD  KEY IS CREREKKEY.

FD TABF.
01 TABFREC.
  03 TABFKEY.
    05 TABFNR                  PIC XXX.
    05 TABFCLE                 PIC X(7).
  03 NO1X                      PIC 9(5).
  03 NO2X                      PIC 9(5).
  03 NO3X                      PIC 9(5).
  ...
FD CREREK.
01 CREREC.
  03 CREREKKEY.
    05 CRKLANT                 PIC X(5).
    05 CRDOKNR                 PIC X(5).
    05 CRVOLG                  PIC 99.
  03 CRPER.
  ...
```

FIGURE 225. The collections and entity types declarations.

```
  MOVE "06 C101" TO TABFKEY.
  READ TABF LOCK INVALID KEY
    STOP RUN.
  IF FILSTAT NOT = "00"
    STOP RUN.
  MOVE NO3X TO NRNU.
  ADD 1 TO NRNU ON SIZE
    ERROR MOVE 0 TO NRNU.
  IF NRNU > NO2X OR NRNU < NO1X
    MOVE NO1X TO NRNU.
  MOVE NRNU TO NO3X.
  REWRITE TABFREC INVALID KEY
    STOP RUN.
  IF FILSTAT NOT = "00"
    STOP RUN.
  UNLOCK TABF RECORDS.
 ****************************
 * ECRITURE DANS LE FICHIER CREREK--
     ****************************
  ACCEPT CRKLANT.
  MOVE NRNU TO CRDOKNR.
  MOVE 0 TO CRVOLG.
...
  WRITE CREREC INVALID KEY
    MOVE "INVALID WRITE CREREK" TO FT1
    GO TO STOPS.
```

FIGURE 226. The procedural fragment.

The code fragment of figure 225 represents the declaration of the two collections and entity types and the code fragment of figure 226 the procedural code fragment. The analysis of the collections and entity types declarations produces the raw physical schema.

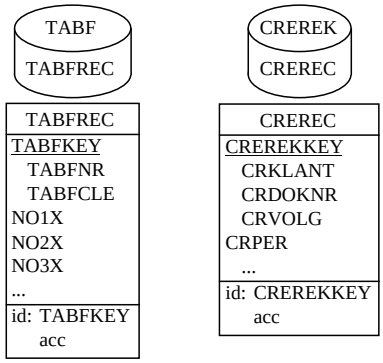


FIGURE 227. The complete physical schema.

3.9.2. Complete physical schema

The analysis of the procedural code (figure 226) shows that the identifier of `TABFREC` is a constant and the value of `NO1X`, `NO2X` and `NO3X` are used to compute the new value of `CRDOKNR` (a part of the identifier of `CREREC`). The value used for `CRDOKNR` replace the value of `NO3X` in `TABFREC`.

This analysis can be interpreted as TABFREC is a technical entity type that does not contains application domain data. This entity type noted as technical through the stereotype <<tech>>. Figure 227 is the complete physical schema.

3.9.3. Conceptual schema

The conceptualization of this schema consist in removing the physical construct (collections and access keys) and in removing the technical entity type (`TABFREC`).

The final conceptual schema is presented in figure 228.

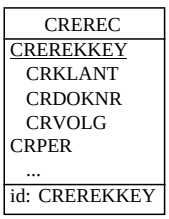


FIGURE 228. The conceptual schema.

C.10. Is-a in SQL

In this example, we have SQL views that represent sub-type of a table. The views are define as follow

```
create view (.....)
as select (.......)
```

```
from <table>
where <column> = <string>;
```

During the conceptualization, we would like to create an is-a relation between the table and all its views.

C.10.1.The complete logical schema

Create a new project and extract (**File/Extract SQL**) the file `is-a.sql`. The result of the extraction is the figure 229. It contains three entity types (*Person*, *Professor*, *Students*).

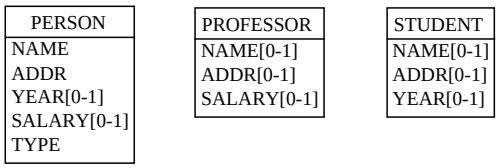


FIGURE 229. The raw physical schema.

The SQL extractor extracts views as entity types and puts the definition of the views into the technical description of the entity type. The tables are extracted as entity types.

So all we have to do is to find the entity types that have a technical description that contains

```
from <table> where <column> = <string>
```

and to create an is-a relation between this entity type and the entity type of name <table>.

The patterns used to search into the technical description are the following.

```
- ::= /g"[/n/t/r ]+";
string ::= /g"'.*'";
name ::= /g"[a-zA-ZO-9_]+";
table ::= name;
column ::= name;
from ::= "from" - @table - "where" - column - "=" - string;
```

The V2 function called to create the is-a relation is the following. The procedure is declared export, because it must be call from outside the voyager program.

```
export procedure create_is-a(string: table)
/* creates a is-a relation between the entity type of name
'table' and the current entity type*/
     data_object : d_obj;
     schema : sch;
     entity_type : sub_ent;
     entity_type : super_ent;
{
     SetPrintList("","","");

     sch := GetCurrentSchema();
/* get the current schema*/
     if IsVoid(sch) then {
/* if there no current schema return an error */
        print("No Schema !\n");
        return;
     }
```

```
        go :=GetCurrentObject();
/* get the current object */
        if IsVoid(go)
        then {
/* if there is no current object, return an error */
          print("No current object !\n");
          return;
        }
        if (GetType(go) <> ENTITY_TYPE)
        then {
/* if the current object is not an entity type, return an error
*/
          print("The current object is not a entity type !\n");
          return;
        }
        sub_ent := go;

/* 'sup_ent' is the entity type of name 'table' */
        sup_ent := GetFirst(DATA_OBJECT[d_obj]{@SCH_DATA:[sch]
             with ((GetType(d_obj) = ENTITY_TYPE)
               and (d_obj.name = table))});

/* 'l_clu' is the list of cluster connected to the super type*/
        l_clu := CLUSTER[clu]{@ENTITY_CLU:[sup_ent]};

        if(Length(l_clu) = 1) then
        {
          /* if the super type has a cluster, use it */
          clu := GetFirst(l_clu);
        }
        else
        {
/* if the super type has no cluster, create it */
          clu := create(CLUSTER, name : sup_ent.name, total : 0,
               disjoint : 0, @ENTITY_CLU : sup_ent);
        }

/* connect the sub-type to the cluster */
        sub_t := create(SUB_TYPE, @CLU_SUB : clu,
               @ENTITY_SUB : sub_ent);
}
```

Now to create the is-a relation select the compact view (to reduce the space research), select the schema and execute the command **Assist - Text analysis - Execute**.

Click on the **OK** button. The is-a relations are created (figure 230).

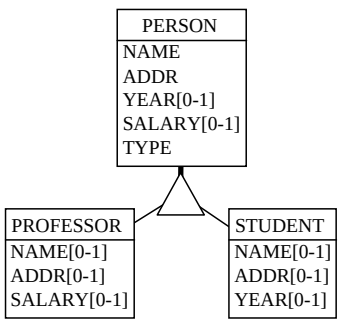


FIGURE 230. The complete logical schema.

C.10.2.Conceptualization

To conceptualize this schema, we have to remove attributes from the super type or the sub-type, according to they are common to both sub-type or not. The result of this conceptualization is the schema of figure 231.

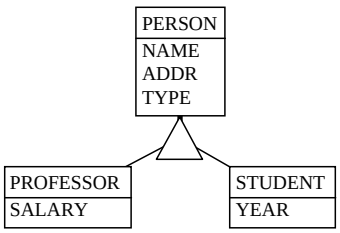


FIGURE 231. The conceptual schema.